VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Logika dynamického diskurzu

Logic of dynamic discourse

2018 Bc. Ivana Kotová

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Ivana Kotová**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Logika dynamického diskursu
Logic of dynamic discourse

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementovat systém pro zpracování dynamického diskursu specifikovaného v jazyce TIL-Script. Vstupem je databáze konstrukcí jazyka TIL-Script. Výstupem bude systém konstrukcí, ve kterých jsou dynamicky doplněny anaforické odkazy na dříve zmíněné objekty.

Práce bude obsahovat:
1. Stručnou definici a popis jazyka TIL-Script.
2. Popis algoritmu pro dynamické zpracování anaforických odkazů.
3. Analýza, návrh a implementace systému pro zpracování dynamického diskursu v jazyce C#.

Seznam doporučené odborné literatury:

[1] Duží M., Jespersen B. and Materna P. (2010): Procedural Semantics for Hyperintensional Logic. Foundations and Applications of Trasnsparent Intensional Logic. First edition. Berlin: Springer, series Logic, Epistemology, and the Unity of Science, vol. 17, ISBN 978-90-481-8811-6.
[2] Duží M., Materna P. (2012): TIL jako procedurální logika (průvodce zvídavého čtenáře Transparentní intensionální logikou). Aleph Bratislava 2012, ISBN 978-80-89491-08-7
[3] Duží M. (2007): Semantic pre-processing of anaphoric references. In RASLAN 2007. Ed. Sojka, P., Horák, A., Brno: Masaryk University, pp. 43-56.
[4] Duží, M. (2017): Logic of Dynamic Discourse; Anaphora Resolution. In Proceedings of the 27th International Conference on Information Modelling and Knowledge Bases - EJC 2017. Thailand

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **prof. RNDr. Marie Duží, CSc.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto diplomovou práci vypracovala samostatně. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

V Ostravě 27. dubna 2018 .....Kolář.....

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.

V Ostravě 27. dubna 2018 .....

Chci na tomto místě poděkovat vedoucí mé diplomové práce profesorce Duží a doktoru Menšíkovi za četné rady a konzultace. Rovněž děkuji svému manželovi a rodině za podporu a trpělivost.

Abstrakt

Cílem této diplomové práce bylo implementovat aplikaci ve formě softwarového agenta, se schopností rozpoznat anaforické odkazy ve větách a vhodně za ně dosazovat tak, abychom získali větu s úplným významem. Text popisuje reprezentaci vět přirozeného jazyka pomocí komputační varianty transparentní intenzionální logiky nazvané TIL-Script. V tomto jazyce jsou věty vyjádřeny konstrukcemi. V rámci práce byl navržen a následně implementován v jazyce C# algoritmus, jenž automaticky buduje databázi z analyzovaných konstrukcí a ty jsou pak substituovány za anaforické odkazy. Tento algoritmus je zasazen v podobě zásuvného modulu do agenta, který je schopen komunikovat s jinými agenty v rámci multi-agentního systému. Modul umožňuje agentu dosazovat vhodné konstrukce na základě kontextu.

Klíčová slova: transparentní intenzionální logika, TIL-Script, logika dynamického diskurzu, anaforické odkazy, agent, multi-agentní systém, ACL, XML

Abstract

The aim of this master thesis was to implement an application in a form of a software agent, which is able to identify anaphoric references in a sentence and substitute for them to obtain a sentence with complete meaning. The theoretical part explains representation of natural language sentences in computational version of transparent intensional logic named TIL-Script. In this logic, sentences are expressed as constructions. An algorithm was designed and then realized in C# language in the implementation part. Algorithm automatically fills a database of analysed constructions. These constructions are afterwards substituted for anaphoric references. The algorithm is injected in a form of a plugin into an agent, which is able to communicate with other agents in a multi-agent system. The plugin enables the agent to substitute an appropriate construction based on the current context.

Key Words: transparent intensional logic, TIL-Script, logic of dynamic discourse, anaphoric references, agent, multi-agent system, ACL, XML

Obsah

Seznam použitých zkratek a symbolů

$\supset$	–	symbol logické spojky implikace
ACL	–	Agent Communication Language (jazyk pro komunikaci agentů)
AJAX	–	Asynchronous JavaScript And XML
API	–	Application Programming Interface
CSRF	–	viz XSRF
CSS	–	Cascading Style Sheets (kaskádové styly)
DI	–	Dependency Injection
EBNF	–	Extended Backus-Naur Form (rozšířená Backus-Naurova forma)
ERD	–	Entity-Relationship Diagram
FIPA	–	Foundation for Intelligent Physical Agents
HTML	–	HyperText Markup Language
HTTP	–	HyperText Transfer Protocol
IDE	–	Integrated Development Environment (integrované vývojové prostředí)
IoT	–	Internet of Things
JSON	–	JavaScript Object Notation
JS	–	JavaScript
LINQ	–	Language Integrated Query
MVC	–	Model-View-Controller
PWS	–	Possible World Semantics (sémantika možných světů)
SW	–	Software
TCP	–	Transmission Control Protocol
TIL	–	Transparentní intenzionální logika
UDP	–	User Datagram Protocol
URI	–	Uniform Resource Identifier
URL	–	Uniform Resource Locator
UTF	–	Unicode Transformation Format
XML	–	eXtensible Markup Language
XSD	–	XML Schema Definition
XSRF	–	Cross-site request forgery

Seznam obrázků

Seznam tabulek

Seznam výpisů zdrojového kódu

1 Úvod

V rámci své diplomové práce se zabývám zpracováním přirozeného jazyka pomocí formálního systému transparentní intenzionální logiky (TIL). Tento logický systém má procedurální sémantiku. To znamená, že analýza výrazu spočívá v nalezení procedury vyjádřené daným výrazem tak, aby bylo možno odvodit co nejvíce relevantních důsledků. TIL je komplexní systém vycházející z typovaného lambda kalkulu s rozvětvenou hierarchií typů. Při analýze se můžeme setkat s výrazy přirozeného jazyka, u nichž není jednoznačný jejich význam. Uvažujme větu: „Ženu holí hnát," v níž může být slovesem kterékoli ze slov. Jelikož lidé při komunikaci intuitivně využívají kontextu, může pro ně být pochopení správného výrazu snadné, narozdíl od agenta, který bez patřičných instrukcí ani nedovede rozpoznat anaforické odkazy[1] ve větách. Například ve výrazu: „Byl jeden král a *ten* měl tři dcery," slovo *král* a *ten* odkazují k téže entitě. Aby byl agent schopen zpracovávat věty s anaforickými odkazy, je zapotřebí navrhnout algoritmus, který odkazy nahradí vhodnými výrazy. Logika dynamického diskurzu v tomto případě řeší substituci vhodného antecedentu[2] za anaforický odkaz *ten* za účelem získání úplného významu věty s anaforickým odkazem.

Cílem práce je navrhnout a implementovat softwarového agenta, jenž umí přijímat zprávy jiných agentů. Z přijatých zpráv pak dovede budovat diskurz antecedentů, o nichž se hovořilo, a v případě, že obdrží zprávu s anaforickým odkazem, provede substituci anafory za vhodný antecedent. Tímto způsobem bude schopen substituovat postupně všechny vhodné antecedenty v diskurzu.

V Kapitole 2 popisuji formální jazyk TIL. Jeho charakteristiky budou vysvětleny na analýze výrazů přirozeného jazyka. Jedná se o nutné teoretické základy, jejichž pochopení je stěžejní pro další části textu. Mimo jiné v této kapitole zmiňuji vztah jazyka TIL ke klasickému lambda kalkulu.

Kapitola 3 je věnována logice dynamického diskurzu. Pod tímto názvem se skrývá schopnost agenta dynamicky doplňovat význam anaforických odkazů k tomu, co bylo dříve řečeno během komunikace s jinými agenty. V TIL lze doplňovat za anaforické odkazy nejen individua, ale konstrukce libovolného typu (např. vlastnosti individuí, propozice, apod.). V kapitole je popsán princip substituční metody, která se při práci s výrazy s anaforickými odkazy používá, včetně analýzy takového výrazu. Rovněž je uveden návrh algoritmu pro řešení anafory a jeho funkce je demonstrována na příkladu.

Následuje Kapitola 4, jež se věnuje komunikaci agentů z pohledu zpráv, které si mezi sebou posílají. První aspekt komunikace je formát analyzovaných výrazů. Pro účely této práce byla zvolena komputační varianta jazyka TIL nazývaná *TIL-Script*. V kapitole bude popsána gramatika jazyka TIL-Script a jeho reprezentace ve formě eXtensible Markup Language (XML)

[1] Anaforický odkaz je typ vnitrotextové reference odkazující k něčemu již řečenému.

[2] Antecedentem je v tomto kontextu myšlen výraz, o němž se hovořilo, a který můžeme substituovat za anaforický odkaz.

struktury. Druhým aspektem je jednotný protokol pro zasílání a příjem zpráv agentů. Na internetu je k dispozici obsáhlá specifikace od společnosti Foundation for Intelligent Physical Agents (FIPA) zabývající se standardizací v oblasti agentů a multi-agentních systémů. Tato specifikace zahrnuje také formát zpráv Agent Communication Language (ACL). Obsahem těchto zpráv budou TIL-Script konstrukce ve formě XML.

Specifikace požadavků a návrh algoritmu jsou popsány v Kapitole 5, kde je uvedeno, jaké vlastnosti systému byly dohodnuty s vedoucí práce a jaké odlišnosti od standardizované specifikace FIPA jazyka (ACL) z těchto vlastností vyplývají. Dále je v kapitole uveden konkrétní návrh implementace algoritmu pro zpracování výrazů s anaforickými odkazy a rovněž řešení komunikace agenta v multi-agentním systému.

Kapitola 6 se zabývá již samotnou implementací algoritmu a reprezentací dat. Jelikož algoritmus pro řešení anafory je naprogramován formou modulu, je zde popsána část systému se zásuvnými moduly. Rovněž se kapitola věnuje práci s databází a logování zpráv z aplikace, síťové komunikaci a uživatelskému rozhraní. Na ACL zprávách je po přijetí provedena lexikální a syntaxtická analýza a výstup z tohoto procesu je předán rozhodovacímu modulu, jenž zvolí k přijaté zprávě vhodnou odpověď. Dále je popsáno, jaký přístup byl zvolen z pohledu testování funkcionality aplikace. Na závěr kapitoly je uveden seznam položek konfigurace aplikace a pokyny pro její nasazení na serveru.

V Kapitole 7 jsou uvedeny dva případy použití aplikace. Během prvního agentu předáváme data, z nichž si tvoří diskurz a také dosazuje za anaforické proměnné. Ve druhém případu je agent dotázán jiným agentem na jména těch agentů, které zná. Zbylá část kapitoly je věnovaná možným vylepšením a rozšířením stávající funkcionality agenta.

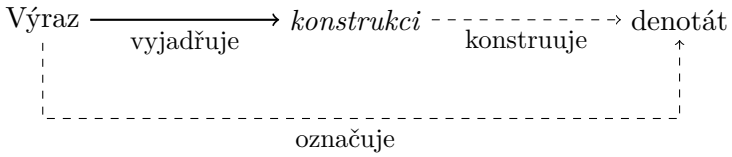
2 Transparentní intenzionální logika

Systém transparentní intenzionální logiky (TIL) je logický systém sloužící převážně k analýze výrazů přirozeného jazyka. Autorem tohoto systému je Pavel Tichý, který jej systematicky popsal ve své knize *The Foundations of Frege's Logic*[1]. Tento logický systém má procedurální sémantiku. To znamená, že analýza výrazu (věty přirozeného jazyka) spočívá v nalezení procedury vyjádřené daným výrazem tak, aby bylo možno odvodit co nejvíce relevantních důsledků.

V této kapitole nejprve zmiňuji základní formální definice systému TIL, v němž se pracuje s rozvětvenou hierarchií typů. TIL lze rovněž chápat jako intenzionální logiku s modálním a temporálním parametrem. Tyto parametry jsou zde také vysvětleny. Následně teoretické základy demonstruji na příkladech analýzy výrazů přirozeného jazyka. TIL vychází z λ-kalkulu, v závěru kapitoly proto zmiňuji společné rysy i odlišnosti, s nimiž jsem se během práce se systémem setkala. Veškeré definice v této kapitole jsou přejaty z knihy *TIL jako procedurální logika* od Marie Duží a Pavla Materny[2].

2.1 Základní formální definice

Sémantiku výrazu zachycuje schéma na Obrázku 1. Hlavním úkolem při analýze výrazu je určení konstrukce, která zachycuje význam výrazu. Taková konstrukce pak konstruuje objekt, který se nazývá denotát. Můžeme se setkat i s výrazy, jež jsou smysluplné, ale denotát nemají, např.: „Poslední desetinná číslice konstanty π“. Jestliže chceme analyzovat výrazy, potřebujeme definovat datové typy, které přiřazujeme označeným objektům.



Obrázek 1: Sémantické schéma analýzy

Definice 1 (typy řádu 1)

Nechť B je báze*, což znamená kolekce vzájemně disjunktních neprázdných množin. Pak:*

i. Každý prvek B je atomický (elementární) typ řádu 1 nad B.

ii. Nechť α, β_1, ..., β_m ($m > 0$) jsou typy řádu 1 nad B. Pak kolekce ($\alpha\beta_1 \dots \beta_m$) všech m-árních parciálních funkcí, tj. zobrazení z kartézského součinu $\beta_1 \times \cdots \times \beta_m$ do α, je molekulární (neboli funkcionální) typ řádu 1 nad B.

iii. Nic jiného není typem řádu 1 nad B *než dle (i) a (ii).*

Při analýze výrazů je základem báze typů řádu 1. Definice 1 připouští pouze dva případy: elementární typy a funkcionální typy. Objekt O typu α nazýváme α-objektem a značíme jej O/α. Tabulka 1 zobrazuje výčet nejčastěji používaných elementárních typů objektové báze.

o	množina pravdivostních hodnot $\{P, N\}$
ι	množina individuí
τ	množina časových okamžiků (nebo také reálných čísel)
ω	množina logicky možných světů

Tabulka 1: TIL: Typy bázových objektů

Ve druhém případě lze objekt chápat jako funkci s argumenty typů $\beta_1, \ldots, \beta_m (m > 0)$, jejíž návratová hodnota je typu α. Například binární relace (=), neboli identita reálných čísel, je $(o\tau\tau)$-objekt. To znamená, že funkce identity vrací pravdivostní hodnotu P pro ty dvojice reálných čísel, kde první je identické s druhým. Jinak funkce vrací N. Jedná se o *extenzionální* objekt, tj. funkci, jejíž doménou není možný svět. Obdobně je to i s dalšími *analytickými* výrazy z oblasti matematiky a logiky, které také označují extenze, např. výrokově-logické spojky $\wedge, \vee, \supset$[3] jsou objekty typu (ooo), $\neg$ objekt typu (oo), apod.

Definice 2 (intenze a extenze)

(α-)intenze *jsou prvky typu $(\alpha\omega)$, tedy funkce z možných světů do libovolného typu α;*

(α-)extenze *jsou objekty typu α, kde $\alpha \neq (\beta\omega)$ pro libovolný typ β; tedy extenze jsou α-objekty jejichž doménou není množina možných světů.*

Pro označení intenzí nejčastěji používáme *empirické* výrazy. Příkladem intenze je výraz „prezident ČR“, jenž neoznačuje konkrétní individuum, nýbrž individuový úřad, tj. objekt typu $((\iota\tau)\omega)$. Dle zavedené konvence je možné zápis typu zkracovat jako $\iota_{\tau\omega}$. Fakt, že určité individuum zastává tento úřad, je z pohledu logiky náhodný. Takový fakt zachycujeme modálním parametrem ω, což je typ množiny možných světů (Possible World Semantics – PWS). Dále temporální parametr τ vyjadřující čas uvádíme, jelikož dané individuum zastávající úřad jej v minulosti nezastávalo a nebude jej zastávat vždy. Některými dalšími často se vyskytujícími intezemi jsou: propozice typu $o_{\tau\omega}$, vlastnosti individuí typu $(o\iota)_{\tau\omega}$ či atributy obecného typu $(\alpha\beta)_{\tau\omega}$, např. „otec (někoho)“, „teplota (něčeho)“, atd.

S dosavadními informacemi uvedenými v tomto textu jsme nyní schopni analyzovat základní typy jednoduchých výrazů. Abychom mohli provádět analýzu složitějších výrazů, vět a souvětí, je zapotřebí zamyslet se nad tím, jak jsou ve větách spojeny objekty základních typů. V TIL jsou definovány *procedury* nebo také *konstrukce*. Na rozdíl od neprocedurálních objektů typů řádu 1 lze konstrukce, jakožto procedury, provést za účelem obdržení nějakého výstupu.

TIL je logika parciálních funkcí. To lze demonstrovat na analyzovaném výrazu „prezident ČR“, ze kterého odvodíme, že v daném světamihu (světě w a čase t) existuje individuum, jež zastává úřad prezidenta ČR. Anebo úřad prezidenta ČR nikdo nezastává a významová konstrukce přiřazená tomuto úřadu je v této valuaci v-nevlastní (používá se rovněž označení *Improper*).

[3] Znak $\supset$ v tomto textu značí funkci implikace a byl zvolen na základě značení použitého v doporučené literatuře.

Můžeme si všimnout, že význam výrazu „prezident ČR“ je složený ze dvou podprocedur. Těmto podprocedurám se říká *konstituenty*.

Konstrukce dělíme na *atomické* a *molekulární*. Atomické konstrukce nemají jiné konstituenty než samy sebe. Jedná se o konstrukce typu *Proměnná* a *Trivializace*. Molekulární konstrukce obsahují kromě sebe i jiné konstituenty. Tyto konstituenty je nejprve nutno provést, než můžeme provést samotnou molekulární konstrukci. Molekulární konstrukce jsou: *Kompozice*, *Uzávěr*, *Provedení* a *Dvojí Provedení*.

Definice 3 (konstrukce)

i. Proměnná x *je* konstrukce*, která konstruuje objekt O příslušného typu v závislosti na valuaci v; tedy x v-konstruuje O.*

ii. Trivializace*: Je-li X jakýkoli objekt (extenze, intenze nebo i* konstrukce*), ${}^0X$ je* konstrukce *zvaná* Trivializace*. Konstruuje objekt X bez jakékoli změny.*

iii. Kompozice $[XY_1 \dots Y_m]$*: Je-li X* konstrukce*, která v-*konstruuje *funkci f typu $(\alpha\beta_1 \dots \beta_m)$, a $Y_1, \dots, Y_m$ v-*konstruují *po řadě objekty $B_1, \dots, B_m$ typů $\beta_1, \dots, \beta_m$, pak tato* Kompozice $[XY_1 \dots Y_m]$ v-konstruuje *hodnotu funkce f na argumentech $B_1, \dots, B_m$ (tj. objekt typu α, pokud f má na $\langle B_1, \dots, B_m \rangle$ hodnotu). Jinak je* Kompozice $[XY_1 \dots Y_m]$ *v-nevlastní, tj. ne* (v-)konstruuje *žádný objekt.*

iv. Uzávěr $[\lambda\, x_1 \dots x_m Y]$ *je* konstrukce*: Nechť $x_1, \dots, x_m$ jsou navzájem různé proměnné, které v-konstruují po řadě objekty typu $\beta_1, \dots, \beta_m$, a nechť Y je konstrukce, která v-konstruuje α-objekt. Pak $[\lambda\, x_1 \dots x_m Y]$* v-konstruuje *funkci $f/(\alpha\beta_1 \dots \beta_m)$, a to takto: Nechť $v(B_1/x_1, \dots, B_m/x_m)$ je valuace, která se liší od valuace v nanejvýš tím, že přiřazuje objekty $B_1/\beta_1, \dots, B_m/\beta_m$ proměnným $x_1, \dots, x_m$. Je-li Y $v(B_1/x_1, \dots, B_m/x_m)$-nevlastní (viz iii), pak funkce f není definována na $\langle B_1, \dots, B_m \rangle$. Jinak je hodnotou funkce f na argumentu $\langle B_1, \dots, B_m \rangle$ α-objekt $v(B_1/x_1, \dots, B_m/x_m)$-konstruovaný Y.*

v. Provedení*: ${}^1X$ je* konstrukce*, která buď v-konstruuje objekt v-konstruovaný konstrukcí X, nebo pokud X není konstrukce nebo je v-nevlastní, je rovněž ${}^1X$ v-nevlastní, tj. nekonstruuje žádný objekt.*

vi. Dvojí Provedení*: ${}^2X$ je* konstrukce*. Tato konstrukce je v-*nevlastní*, pokud X není konstrukce, nebo pokud X ne v-konstruuje jinou konstrukci, nebo v-konstruuje v-nevlastní konstrukci. Jinak, jestliže X v-konstruuje konstrukci Y a Y v-konstruuje objekt Z, pak ${}^2X$ v-konstruuje Z.*

vii. Nic jiného není konstrukce *než dle (i) – (vi).*

Pro zjednodušení formálního zápisu jsou ustáleny následující konvence. Vnější závorky výrazu je možné vypustit tam, kde nemůže dojít k pochybení, zejména u konstrukce typu Uzávěr.

Tak například místo $[\lambda\, x\, y[^0{=}\ x\ y]]$ lze psát $\lambda\, x\, y[^0{=}\ x\ y]$. Logické a matematické operátory je možno zapisovat infixní notací bez trivializace. V uvedeném příkladě bude konstrukce identity proměnných x a y vypadat takto: $\lambda\, x\, y[x = y]$. Slovní obrat v-konstruuje je zkrácením fráze „konstruuje v dané valuaci v“. Dále v textu toto vyjadřuji zápisem $x \rightarrow_v \iota$, což de facto znamená proměnná x v-konstruuje objekt typu ι. Další často používanou zkratkou v zápisu je uvádění temporální a modální proměnné v dolním indexu při extenzionalizaci daného intenzionálního objektu. Ku příkladu výraz „Pianista“ označuje vlastnost individua „být pianistou“, což zapisujeme jako $Pianista/(o\iota)_{\tau\omega}$. V tomto případě výraz neoznačuje množinu individuí, tj. extenzi typu $(o\iota)$, ale specifikuje ta individua, která jsou v daném světě w a časovém okamžiku t pianisty. U konstrukcí konstruujících intenzi pak často se vyskytující aplikaci této funkce na modální a temporální parametry, tj. $[[Pianista\, w]\, t]$, zkracujeme zápisem $Pianista_{wt}$.

Uvažujme nyní analýzu složitějšího výrazu: „Albert zkoumá závislost mezi energií a hmotou“. Se základní Definicí 1 si již nevystačíme. Budeme potřebovat konstrukce a to nejen pro vyjádření abstrakce daného světačasu, ale také k popisu vztahu mezi zkoumajícím individuem a předmětem zkoumání. Analyzujeme-li jednotlivé objekty, pak narazíme na problém. K čemu má v tomto příkladu Albert vztah? Jistě ne ke konkrétní pravdivostní hodnotě vyjadřující rovnost energie a hmoty krát rychlost světla na druhou (tedy extenzi $E = mc^2$) — kdyby věděl, že je to pravda, už by přeci nemusel nic zkoumat. V tomto případě se jedná o vztah k významu výrazu, tedy ke konstrukci. Analýzu konstrukcí provádíme pomocí rozvětvené hierarchie typů. Konstrukce konstruující objekt základního typu (řádu 1) se označují $*_1$. Konstrukce řádu 1 jsou rovněž objekty typu řádu 2.

Definice 4 (rozvětvená hierarchie typů nad bází B)

1. T_1 (typy řádu 1) *byly definovány v Definici 1.*

2. C_n (konstrukce řádu n)

   i. *Nechť x je proměnná, která v-konstruuje objekty typu řádu n. Pak x je* konstrukce řádu n nad B.

   ii. *Nechť X je prvek typu řádu n. Pak ${}^0X, {}^1X, {}^2X$ jsou* konstrukce řádu n nad B.

   iii. *Nechť $X, X_1, \ldots, X_m (m > 0)$ jsou konstrukce řádu n nad B. Pak $[XX_1 \ldots X_m]$ je* konstrukce řádu n nad B.

   iv. *Nechť $x_1, \ldots, x_m, X (m > 0)$ jsou konstrukce řádu n nad B. Pak $[\lambda\, x_1 \ldots x_m\, X]$ je* konstrukce řádu n nad B.

   v. *Nic jiného není* konstrukce řádu n nad B *než to, co je definováno dle C_n (i)-(iv).*

3. T_{n+1} (typy řádu $n+1$)
   Nechť $*_n$ je kolekce všech konstrukcí řádu n nad B.

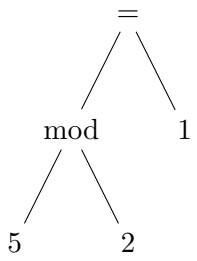
   i. *$*_n$ a každý typ řádu n jsou* typy řádu $n+1$ nad B.

ii. Jsou-li $\alpha, \beta_1, \ldots, \beta_m (m > 0)$ *typy řádu* $n+1$ *nad* B*, pak* $(\alpha\beta_1 \ldots \beta_m)$*, tj. kolekce parciálních funkcí – viz* T_1 *(ii), je* typ řádu $n+1$ nad B.

iii. Nic jiného není typ řádu $n+1$ nad B *než dle* T_{n+1} *(i) a (ii).*

2.2 Analýza výrazů

Nyní již máme k dispozici potřebné definice, abychom mohli začít s analýzou výrazů přirozeného jazyka. Vezměme si ku příkladu jednoduchý matematický výraz „5 mod 2 = 1“. Na první pohled je zřejmé, že vyhodnotíme-li celý výraz, získáme pravdivostní hodnotu P. Dále si můžeme všimnout, že se celý výraz skládá z podvýrazů — levá a pravá strana relace rovnosti. Podvýraz na levé straně pak dále můžeme rozložit na objekty atomického typu (τ). Rozklad výrazu na podvýrazy přehledně vyjádříme pomocí stromu na Obrázku 2.



Obrázek 2: Rozklad výrazu „5 mod 2 = 1“ na podvýrazy

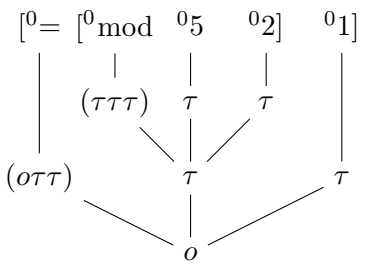
Nyní provedeme typovou analýzu objektů. Začněme s objekty atomických typů v listech a pokračujme směrem vzhůru k objektům molekulárního typu až po kořen. Čísla 5, 2 i 1 jsou (τ)-objekty, tedy značíme $5, 2, 1/\tau$. Operace modulo je funkce, která pro dvě čísla (argumenty funkce) vrací zbytek po dělení prvního čísla druhým, tedy $\mathrm{mod}/(\tau\tau\tau)$. Konečně relaci rovnosti analyzujeme v tomto výrazu jako $=/o\tau\tau$. Samotný výraz pak označuje objekt typu o.

Když je hotov výčet datových typů, pokusíme se je vhodně složit (syntetizovat) do konstrukcí tak, abychom analyzovali daný výraz. Pomoci si můžeme pohledem na strom rozkladu a ačkoli zavedené konvence umožňují zápis matematických operací infixní notací bez trivializace, v tomto případě, pro lepší přehlednost, analyzujeme výraz klasickým způsobem.

$$[^0= [^0\mathrm{mod}\ \ ^0 5\ \ ^0 2]\, ^0 1] \tag{1}$$

Jakmile máme hotovu syntézu, provedeme kontrolu typů. Podobně jako při analýze je vhodné začít od objektů atomických typů, jež jsou zároveň argumenty, na něž aplikujeme funkci. Návratové hodnoty z funkcí nám pak dají typ argumentu, na který můžeme aplikovat další funkce. Datový typ konečné návratové hodnoty by se měl shodovat s typem samotného výrazu, o čemž se lze přesvědčit na Obrázku 3.

Vypsání typů předem je konvence, kterou je vhodné dodržovat pro udržení přehlednosti ve výčtu všech typů jednotlivých podvýrazů a pro možnost se k nim snadno vracet v případě



Obrázek 3: Typová kontrola analýzy výrazu „5 mod 2 = 1“

potřeby. Zejména na počátku studia problematiky se mi osvědčilo vždy provádět typovou kontrolu, dokud nezískám potřebné zkušenosti s typovou analýzou a syntézou.

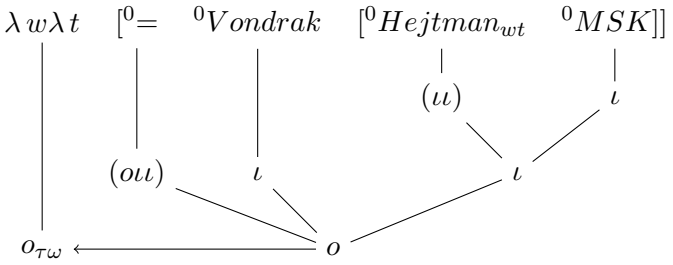
Nyní budeme analyzovat jiný příklad: „Vondrák je hejtmanem Moravskoslezského kraje“. Začneme samozřejmě typovou analýzou. Výraz Vondrák označuje objekt typu individuum, tedy $Vondrak/\iota$. Dále Moravskoslezský kraj pro zjednodušení formálního zápisu vyjádříme zkratkou MSK a analyzujeme jako MSK/ι. Slovo hejtman (něčeho) vyjadřuje empirickou funkci (intenzi) a analyzujeme jej $Hejtman/(\iota\iota)_{\tau\omega}$. Dále výraz „hejtman MSK“ označuje individuový úřad, tj. objekt typu $\iota_{\tau\omega}$. To znamená, že v daném světě, čase a kraji je pouze jedno individuum, zastávající tuto funkci (pokud vůbec nějaké). Operace identity individuí je pak typu $=/(o\iota\iota)$ a jelikož očekává dva argumenty typu ι, je potřeba individuový úřad „hejtman MSK“ (funkci) extenzionalizovat aplikací funkce na modální a temporální argumenty typu $\tau\omega$. Celý výraz pak vyjadřuje propozici v intenzi, což je datový typ $o_{\tau\omega}$. Syntéza výrazu může vypadat následovně:

$$\lambda w \lambda t\, [^0= \ ^0Vondrak \ \lambda w_1 \lambda t_1\, [^0Hejtman_{w_1 t_1} \ ^0MSK]_{wt}] \tag{2}$$

Jelikož TIL je systém založený na typovém lambda kalkulu, můžeme si v tomto případě analýzu zjednodušit aplikováním pravidla β-redukce. Následující konstrukce vznikla aplikováním pravidla nejprve pro λw_1 a poté λt_1. Podrobná analýza β-redukce je uvedena v Podkapitole 2.3.

$$\lambda w \lambda t\, [^0= \ ^0Vondrak \ [^0Hejtman_{wt} \ ^0MSK]] \tag{3}$$

Nakonec provedeme typovou kontrolu, která je znázorněna na Obrázku 4.



Obrázek 4: Typová kontrola analýzy věty „Vondrák je hejtmanem Moravskoslezského kraje“

Na tomto místě je vhodné zmínit, že při analýze výrazu je možné zvolit různou úroveň podrobností, které budou do analýzy výrazu zahrnuty. Například v posledním příkladu lze podvýraz „hejtman Moravskoslezského kraje“ rovnou analyzovat jako individuový úřad typu $\iota_{\tau\omega}$ bez nutnosti analyzovat zvlášť empirickou funkci (intenzi) *hejtman (něčeho)* a zvlášť *MSK* typu individuum. Individuový úřad bychom si pak mohli označit např. zkratkou *HMSK* (hejtman Moravskoslezského kraje). Při zvažování míry granularity analýzy je dobré si uvědomit, že hrubší přístup může usnadnit práci s analýzou, nicméně omezuje množství informací z analýzy výrazu odvoditelných. Naopak jemnější úroveň formální analýzy může být značně pracná a pod určitou úroveň již nepřináší žádné nové poznatky.

Shrneme-li nabyté znalosti, pak jsme se naučili analýzu provádět ve třech krocích:

1. *Typová analýza objektů*, o kterých daný výraz V hovoří, tj. těch objektů, které jsou označeny podvýrazy výrazu V se samostatným významem.

2. *Syntéza*, „poskládání“ konstrukcí objektů ad (1) tak, aby bylo dosaženo konstrukce objektu označeného výrazem V.

3. *Typová kontrola*, tedy kontrola, zda byla syntéza provedena v souladu s typovými pravidly vyplývajícími z Definice 3.

2.3 TIL jako lambda kalkul

Jak již bylo zmíněno, systém TIL vychází z typového λ-kalkulu [3]. Typový λ-kalkul nabízí tři základní prvky: proměnné, abstrakce a aplikace, kde každý prvek má přiřazen konkrétní typ. V TIL těmto prvkům odpovídají konstrukce Proměnné, Uzávěr a Kompozice, a rovněž v něm lze modifikovat výrazy dle platných pravidel λ-kalkulu za použití α-redukce, β-redukce a η-redukce (uvádí se také označení -konverze). Částečně jsme se o tom mohli přesvědčit u analýzy výrazu v předchozí části této kapitoly, kde byla aplikována β-redukce.

Klasický lambda kalkul používá pouze jednoargumentové funkce. To znamená, že funkce, která vyžaduje dva argumenty (případně více argumentů), např. funkce sčítání, je přetvořena do ekvivalentní funkce vyžadující jediný argument. Na výstupu tato funkce vrací další funkci, jež vyžaduje také jediný argument. Výstupem druhé funkce je výsledná hodnota. Toto skládání se v literatuře nazývá *currying*. Při β-redukci se provádí substituce lambda vázané proměnné za argument výrazu v těle abstrakce. V notaci lambda kalkulu to můžeme zapsat takto: $((\lambda x.M)E) \rightarrow (M[x := E])$, kde x je proměnná, M lambda term a E výraz argumentu. V klasickém λ-kalkulu je implicitně použito uzávorkování zleva doprava, pokud není explicitně uvedeno jinak. Přičemž každá funkce má pouze jediný argument. β-redukce je krok, ve kterém se provede aplikace pro jedinou lambda funkci.[4]

Narozdíl od tohoto přístupu TIL pracuje i s víceargumentovými funkcemi. Vezměme už samotnou definici kompozice (viz Definice 3), která umožňuje skládat funkci s libovolným pře-

dem daným počtem argumentů. Tento rozdíl je patrný také při aplikaci β-redukce. Pro úplnost na tomto místě uveďme definici beta transformace pro TIL.

Definice 5 (β-transformace)

Nechť $x_i \rightarrow_v \alpha_i$ $(1 \leq i \leq m)$ *jsou navzájem různé proměnné a* $D_i \rightarrow_v \alpha_i$ *konstrukce. Dále nechť* $Y(D_i/x_i)$ *je konstrukce, která vznikne z konstrukce* Y *korektní substitucí konstrukcí* D_i *za všechny výskyty proměnné* x_i $(1 \leq i \leq m)$ *v konstrukci* Y*. Pak přechod*

$$[[\lambda x_1 \dots x_m Y] D_1 \dots D_m] \vdash Y(D_i/x_i)$$

nazýváme pravidlo β-redukce *a přechod*

$$Y(D_i/x_i) \vdash [[\lambda x_1 \dots x_m Y] D_1 \dots D_m]$$

navýváme pravidlo β-rozvinutí.

Vezměme si ku příkladu znovu analyzovaný výraz „Vondrák je hejtmanem Moravskoslezského kraje“ před aplikací β-redukce a pro názornost nyní nepoužijeme zkracování zápisu.

$$\lambda w \lambda t\, [^0= \ ^0Vondrak\ [[[\lambda w_1 \lambda t_1\, [[[^0hejtman\, w_1]\, t_1]\ \ ^0MSK]]\, w]\, t]] \tag{4}$$

Jako pomůcku si rovněž vytvoříme abstraktní syntaktický strom výrazu λ-kalkulu znázorněný na Obrázku 5. Aplikaci β-redukce můžeme v TIL provést pouze v případě, kdy máme k dispozici konstrukci Uzávěru nějaké funkce f (lambda abstrakci) a konstrukci argumentu funkce f. V uvedeném příkladu jsou dvě lambda abstrakce s argumentem. Nejedná se tedy o lambda abstrakci s více argumenty. Můžeme postupovat stejně jako v klasickém lambda kalkulu a to tak, že ve dvou krocích provedeme aplikaci lambda funkce na argument. První lambda, u které můžeme aplikovat β-redukci je λw_1. Aplikací dosadíme za všechny výskyty w_1 v lambda termu konstrukci argumentu příslušného datového typu, v tomto případě typu ω. Argument příslušného datového typu ω je konstruován proměnnou w. V syntaktickém stromu je tento argument dané lambda proměnné nejblíže. Aplikací získáme následující konstrukci.

$$\lambda w \lambda t\, [^0= \ ^0Vondrak\ [[\lambda t_1\, [[[^0hejtman\, w]\, t_1]\ \ ^0MSK]]\, t]] \tag{5}$$

Obdobně ve druhém kroku aplikací β-redukce dosadíme za všechny výskyty t_1 v lambda termu konstrukci argumentu t objektu typu τ. Výsledek je uveden jako konstrukce 3 v předchozí Podkapitole 2.2.

Pokud se v lambda termu TIL výrazu nachází lambda s více proměnnými, dosazují se všechny argumenty současně. β-redukce obou proměnných se pak provádí v jediném kroku. Jako příklad uvažujme následující transformaci:

$$[[\lambda x y\, [^0- x\, y]]\ ^0 8\ ^0 5] \Rightarrow [^0- \ ^0 8\ \ ^0 5] \tag{6}$$

Jelikož se jednalo o jedinou dvouargumentovou funkci, mohli jsme dosadit oba argumenty a převést rovnou do výsledného tvaru. Toto lze provést pouze v případě, kdy máme shodný počet lambda proměnných a argumentů a zároveň se pro každou dvojici shoduje typ. Pokud nesedí počet argumentů nebo jejich typy, není pravidlo β-redukce aplikovatelné. Proto je třeba dbát zvláštní obezřetnosti při aplikaci tohoto pravidla.

Pravidlo β-redukce jménem není v TIL obecně použitelné také proto, že se jedná o logiku parciálních funkcí. Mějme následující konstrukce:

$$[\lambda x\, \lambda y\, [y - x]]\ [^0\ln\ ^0{-}1] \tag{7}$$

$$\lambda y\, [y - [^0\ln\ ^0{-}1]] \tag{8}$$

Určeme si datové typy: $x, y \to \tau$, operátor odčítání $-/(\tau\tau\tau)$, funkce přirozeného logaritmu $\ln/(\tau\tau)$, a celé číslo $-1/\tau$.

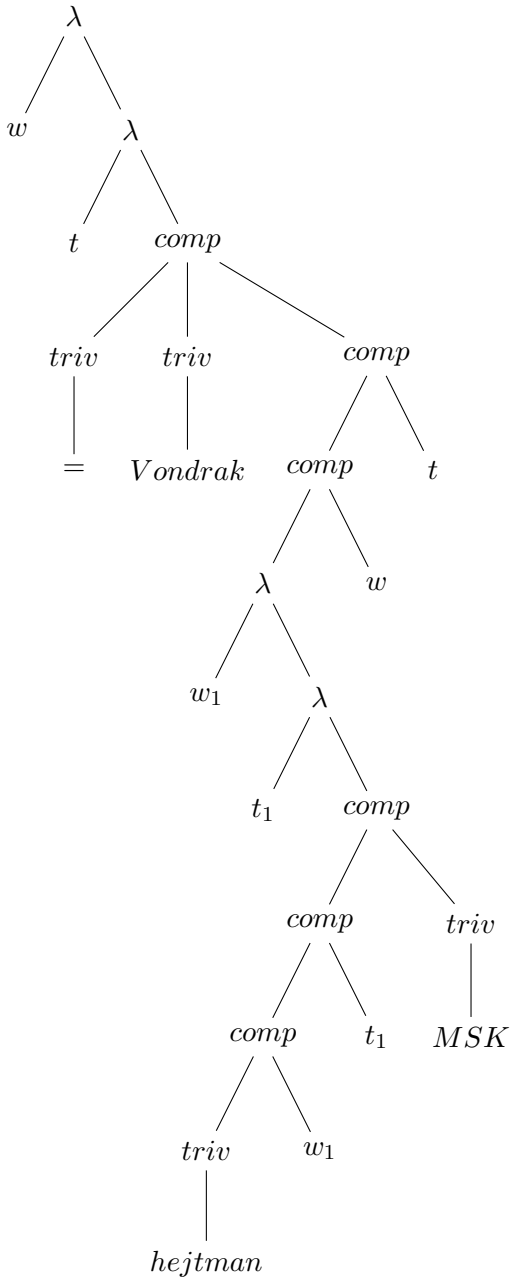
Přirozený logaritmus není na argumentu -1 definován v oboru reálných čísel. Konstrukce argumentu v konstrukci 7 je v tomto případě *nevlastní* (nekonstruuje nic). Z principu kompozicionality vyplývá, že celá konstrukce je také nevlastní, jelikož lambda abstrakce neobdrží argument, na nějž by ji bylo možné aplikovat. Oproti tomu konstrukce 8 konstruuje funkci, která je nedefinovaná na každém svém argumentu. Takovou funkci nazýváme *degenerovanou*. Jak jsme zjistili, tyto konstrukce nejsou ekvivalentní.

V TIL rozlišujeme dva typy aplikace β-redukce. Výše uvedená definice je prvním typem a tím je aplikace *jménem*. To znamená, že za proměnnou dosazujeme samotnou konstrukci argumentu. Tato konstrukce je tedy provedena až po dosazení.

Obdobného přístupu si můžeme všimnout také v některých funkcionálních jazycích (např. Haskell), které adoptovaly tzv. *lazy evaluation* přístup. Argumenty funkce jsou pak vyhodnoceny až ve chvíli, kdy je to nezbytné pro další běh programu (pokud vůbec). Díky tomu lze pracovat s nekonečnými nebo částečně definovanými strukturami. [5].

Druhým typem β-redukce je aplikace *hodnotou*, při které se konstrukce nejprve provede a teprve výsledek provedení (pokud funkce vrací nějakou hodnotu) nahradíme za všechny výskyty proměnné. O tomto typu je znovu zmínka v následující kapitole, kdy můžeme využít substituční metody jako ekvivalentní úpravy β-redukce hodnotou, při které nedochází k nežádoucí ztrátě analytické informace. Tato úprava také zaručuje zachování ekvivalence.

Na závěr této kapitoly ještě zmíníme, že v příkladu analýzy výrazu „Vondrák je hejtmanem Moravskoslezského kraje“ byla použita *redukovaná* nebo také *omezená* verze β-redukce (v angličtině *restrictive*). Při této transformaci provádíme substituci pouze konstukcí typu proměnná nebo atomické trivializace, které nemohou být nevlastní. Při použití je výsledná konstrukce vždy ekvivalentní s původní a nejedná se v tomto případě v pravém slova smyslu o proceduru aplikace funkce na argument, ale jen o jakési technické zjednodušení dané konstrukce. Více o tomto tématu například ve článku [6].



Obrázek 5: Abstraktní syntaktický strom výrazu λ-kalkulu „Vondrák je hejtmanem Moravskoslezského kraje“

Zkratka *comp* značí, že uzel na dané úrovni je Kompozice. Potomci bezprostředně následující z uzlu jsou jednotlivé složky Kompozice, tedy nějaká funkce a její argumenty. Dále zkratka *triv* označuje Trivializaci. Uzly Trivializace mají vždy pouze jednoho potomka – atomickou či elementární konstrukci. Poslední uvedenou zkratkou je λ. Takto označený vrchol je Uzávěrem. Potomci uzávěru jsou lambda proměnné a lambda funkce.

3 Logika dynamického diskurzu

Zabýváme-li se analýzou přirozeného jazyka nějakou dobu, nezbytně musíme dojít do situace, kdy si uvědomíme, nakolik je naše vyjadřování nejednoznačné. V úvodu jsem například zmínila větu: „Ženu holí hnát," v níž může být slovesem kterékoli ze slov. Lidé při komunikaci intuitivně využívají kontextu, ve kterém hovoří bez nutnosti explicitně vyjadřovat stále dokola, že je řeč o konkrétním denotátu. V úvodu byla také zmíněna věta „Byl jeden král a *ten* měl tři dcery." Přestože v tomto příkladu není namísto zájmena *ten* opakováno slovo *král*, jsme schopni si kontext domyslet. Stroj toto však nedovede bez potřebných instrukcí. Aby mohl automatizovaně zpracovávat věty s anaforickými odkazy, je zapotřebí použít algoritmus, který odkazy nahradí vhodnými antecedenty. Tímto problémem se zabývá logika dynamického diskurzu — metoda pro výpočet úplného významu vět s anaforickými odkazy. Díky tomu, že TIL je hyperintenzionální typovaný lambda kalkul, je možné k anaforickým proměnným navázat nejen individua, ale i entity libovolného typu: vlastnosti individuí, propozice, konstrukce, atd.

V textu této kapitoly jsou vysvětleny aspekty TIL potřebné pro pochopení principu substituční metody a analýzy výrazů s anaforickými odkazy. Veškeré definice v této kapitole jsou přejaty z knihy *TIL jako procedurální logika* od Marie Duží a Pavla Materny[2]. Také je v této kapitole uveden algoritmus navržený v článku Marie Duží [7] pro zpracování anaforických odkazů.

3.1 Volné a vázané proměnné

Při použití proměnné v konstrukci je třeba rozlišovat její volné a vázané výskyty. Například v konstrukci $[[^0 5 \text{ mod } x] = {}^0 1]$ se proměnná x vyskytuje volně. Proměnné lze v TIL vázat dvěma způsoby: pomocí konstrukce lambda abstrakce (čili Uzávěru) „λ" nebo Trivializací „^{0}". Takto vypadá λ-vázaná proměnná x:

$$\lambda x\, [[^0 5 \text{ mod } x] = {}^0 1] \tag{9}$$

Proměnná x má v tomto výrazu pouze jediný výskyt. Označení $\lambda\, x$ značí operaci abstrakce od hodnot proměnné, nikoli další výskyt proměnné x. Konstrukce abstrahuje od hodnoty proměnné x, konstruuje celou funkci, která vrací všechny takové prvky, jimiž je číslo 5 dělitelné se zbytkem 1. Může se také stát, že konstrukce C neobsahuje žádný výskyt proměnné x, značíme $[\lambda x\, C]$. Například uzávěr $[\lambda x\, [^0 - {}^0 8 \;\; {}^0 5]]$, v němž proměnná $x \to_v \tau$, konstruuje konstantní funkci typu $(\tau\tau)$, jež libovolnému číslu přiřazuje číslo 3.

Druhým případem vazby je ^{0}-vázaná proměnná, např. proměnná x je vázaná Trivializací v následující konstrukci:

$$^0[\lambda x\, [[^0 5 \text{ mod } x] = {}^0 1]] \tag{10}$$

Tato konstrukce konstruuje uzávěr $[\lambda x\,[[{}^0 5 \text{ mod } x] = {}^0 1]]$. Vázání Trivializací je silnější než vázání operátorem λ.

Definice 6 (volné/vázané proměnné, otevřené/uzavřené konstrukce)
Nechť C je konstrukce s alespoň jedním výskytem proměnné ξ.

i. Nechť C je ξ. Pak výskyt *ξ v C je* volný.

ii. Nechť C je identické s konstrukcí ${}^0X$. Pak každý výskyt *ξ v C je* ^{0}-vázaný *(vázaný trivializací).*

iii. Nechť C je $[\lambda x_1 \ldots x_n\, Y]$ a nechť ξ je identická s některou z proměnných $x_i\,(1 \leq i \leq n)$. Pak každý výskyt *ξ v Y je* λ-vázaný *v C, pokud není* ^{0}-vázaný *v Y. Není-li ξ identická s některou z proměnných $x_i\,(1 \leq i \leq n)$, a její* výskyt *není* ^{0}-vázaný *nebo* λ-vázaný *v Y, pak je* výskyt *ξ* volný *v C.*

iv. Nechť C je $[XX_1 \ldots X_n]$. Pak libovolný výskyt *ξ, který je* volný, ^{0}-vázaný, λ-vázaný *v některé z konstrukcí $X, X_1, \ldots, X_n$, je* volný, ^{0}-vázaný, λ-vázaný *v C.*

v. Nechť C je ${}^1X$. Pak libovolný výskyt *ξ, který je* volný, ^{0}-vázaný, λ-vázaný *v X, je* volný, ^{0}-vázaný, λ-vázaný *v C.*

vi. Nechť C je ${}^2X$. Pak libovolný výskyt *ξ, který je* volný, ^{0}-vázaný, λ-vázaný *v některém konstituentu C, je* volný, ^{0}-vázaný, λ-vázaný *v C. Je-li* výskyt *ξ* ^{0}-vázaný *v konstituentu ${}^0D$ konstrukce C a tento* výskyt *D je konstituentem konstrukce X' v-konstruované konstrukcí X, pak je-li* výskyt *ξ* volný *nebo* λ-vázaný *v D, je tento* výskyt volný *nebo* λ-vázaný *také v C. Jinak je* výskyt *ξ* ^{0}-vázaný *v C.*

vii. výskyt *ξ je* volný, λ-vázaný, ^{0}-vázaný *v C pouze dle (i)-(vi).*

Konstrukce s alespoň jedním výskytem volné proměnné je otevřená konstrukce. *Konstrukce bez výskytu volných proměnných je* uzavřená konstrukce.

Vázané proměnné mají tu nevýhodu, že za ně není možné přímo substituovat. Naštěstí TIL jako hyperintenzionální kalkul může pracovat s konstrukcemi jakožto objekty a není problém v něm uvolnit proměnnou v hyperintenzionálním kontextu — vázanou trivializací. Za tímto účelem se používají funkce substituce Sub_n a trivializace Tr_α.

3.2 Funkce trivializace a substituční metoda

Funkce trivializace $Tr_\alpha/(*_n\alpha)$ pro prvek a typu α (argument) vrací trivializaci tohoto prvku. Ku příkladu $[{}^0Tr_\tau\ {}^0 5]$ vrací konstrukci ${}^0 5$, narozdíl od konstrukce Trivializace, která např. pro konstrukci ${}^0 5$ vrací číslo 5. Zatímco konstrukce Trivializace ${}^0x$ váže proměnnou x, je tato proměnná volná v Kompozici $[{}^0Tr_\tau\ x]$. Dolní index funkce trivializace se musí shodovat s datovým typem argumentu. Nemůže-li dojít ke sporu, může být dolní index funkce trivializace vynechán.

Substituční metoda využívá funkci $Sub_n/(*_n\ *_n\ *_n)$, jež vrací konstrukci n-tého řádu, získanou aplikací argumentů X, Y a Z typu $*_n$ následovně: dosaď X za Y do Z.

Příklad bude jistě hovořit za vše. Uvažujme výraz „Existuje číslo, na kterém není funkce tangens definována.“ Z matematické teorie víme, že takové číslo existuje, je to $\frac{\pi}{2}$. Pokusíme-li se o analýzu věty, zjistíme, že jediný výskyt proměnné x je v konstituentu ${}^0[{}^0\mathrm{tg}\, x]$ vázaný trivializací a tudíž nedostupný pro operátor λ. Následující konstrukce nevznikla korektní analýzou věty:

$$ {}^0\exists\lambda x\,[{}^0Improper\ {}^0[{}^0\mathrm{tg}\, x]] \tag{11}$$

Typová analýza objektů: $\exists/(o(o\tau))$, $x \rightarrow_v \tau$, funkce tangens $\mathrm{tg}\,/(\tau\tau)$ a $Improper/(o*_1)$: třída konstrukcí v-nevlastních pro každou valuaci v. To znamená, že na množině všech v-nevlastních konstrukcí nám funkce vrátí pravdivostní hodnotu, zda daná konstrukce je či není prvkem této množiny. Podrobnější definice existenčního kvantifikátoru je k dispozici v monografii [2] (Definice 2.10 (kvantifikátory), strana 70).

Abychom však mohli provést korektní syntézu, je zapotřebí osvobodit proměnnou x z hyperintenzionálního kontextu. Toho docílíme aplikací substituční metody a funkce trivializace. V následujícím výrazu je proměnná x λ-vázaná, tzn. nepodléhá trivializaci konstrukce ${}^0[{}^0\mathrm{tg}\, x]$.

$$ {}^0\exists\lambda x\,[{}^0Improper\,[{}^0Sub\,[{}^0Tr\, x]\,{}^0x\,{}^0[{}^0\mathrm{tg}\, x]]] \tag{12}$$

Třída čísel konstruovaná tímto Uzávěrem je neprázdná, protože obsahuje číslo $\frac{\pi}{2}$. Tím pádem existenční generalizace konstruuje pravdivostní hodnotu P.

Substituční metodu lze také použít jako ekvivalentní úpravu pro pravidlo β-redukce hodnotou. Při tomto kroku nedochází ke ztrátě analytické informace a k zachování ekvivalence mezi konstrukcemi. V předchozí kapitole jsme se seznámili s β-redukcí jménem. Podívejme se na výraz $[[\lambda xy\,[{}^0{-}\,x\ \ y]]\ {}^0{8}\ {}^0{5}] \rightarrow [{}^0{-}\ {}^0{8}\ \ {}^0{5}]$. Nyní si představme, že nemáme k dispozici původní tvar konstrukce před transformací. V takové situaci není možné z výsledné konstrukce určit, jak jsme k ní dospěli (tzn. jak vypadala konstrukce, která ji konstruovala). Definice substituční metody toto umožňuje. Zde je příklad, který to demonstruje. Mějme původní konstrukci:

$$ [[\lambda xy\,[{}^0{-}\,x\ \ y]]\ {}^0{8}\ {}^0{5}] \tag{13}$$

Nyní definujme funkci substituce. Jelikož funkce je dvouargumentová, při analýze vznikly dvě zanořené funkce substituce. Všiměme si, že v prvním kroku je stále k dispozici informace, jaká funkce byla aplikovaná na jaký argument.

$$ {}^2[{}^0Sub\ {}^{00}8\ {}^0x\,[{}^0Sub\ {}^{00}5\ {}^0y\ {}^0[{}^0{-}\,x\ \ y]]] = {}^{20}[{}^0{-}\ {}^0{8}\ \ {}^0{5}] = [{}^0{-}\ {}^0{8}\ \ {}^0{5}] \tag{14}$$

Místo funkce trivializace jsme použili zkrácený zápis: ${}^{00}8$. Rovněž stojí za zmínku složení značek ^{20}, s nímž jsme se setkali již u Definice 6 ve druhé části bodu vi). Dvojí provedení vyruší jednu operaci trivializace. Předposlední a poslední konstrukce tedy konstruují tentýž objekt,

jsou ekvivalentní. Pokud by se mezi konstituenty nacházela proměnná, nejednalo by se v tomto případě o ^{0}-vázanou proměnnou, nýbrž o volnou proměnnou. Definice 7 shrnuje definici β-redukce hodnotou.

Definice 7 (β-redukce hodnotou)

Nechť pro $1 \leq i \leq m$ *jsou* $x_i \rightarrow_v \alpha_i$ *navzájem různé proměnné a* $D_i \rightarrow_v \alpha_i$ *konstrukce. Pak přechod*

$$[[\lambda x_1 \ldots x_m\, Y]\, D_1 \ldots D_m] \vdash {}^2[{}^0Sub\, [{}^0Tr_{\alpha_1}\, D_1]\, {}^0x_1 \ldots [{}^0Sub\, [{}^0Tr_{\alpha_m}\, D_m]\, {}^0x_m\, {}^0Y]]$$

nazveme pravidlo β-redukce hodnotou.

Téma substituční metody si zajisté zaslouží naši další pozornost. Vždyť se jedná o hlavní stavební kámen, který se používá při práci s anaforickými proměnnými. Pro ujištění, že jsme s látkou dobře srozuměni, je zde další příklad tentokrát v intenzionálním kontextu. Je dána věta: „Karel počítá 5 mod $x = 1$." Z podobného příkladu v předchozím textu můžeme již předem odvodit, že ve výrazu má individuum Karel vztah ke konstrukci. Analýza tedy vypadá následovně: $Karel/\iota$, $pocitat/(o\iota{*_1})_{\tau\omega}$, $5, 1/\tau$, $\mathrm{mod}/(\tau\tau\tau)$, $=/(o\tau\tau)$, $x \rightarrow_v \tau$, tedy proměnná x v-konstruuje objekt typu τ, 5 mod $x = 1/(*_1)$ je konstrukce řádu 1. Nyní provedeme syntézu výrazu:

$$\lambda w \lambda t\, [{}^0Pocitat_{wt}\ {}^0Karel\ {}^0[[{}^0 5 \bmod x] = {}^0 1]] \tag{15}$$

Zde bychom už věděli jak postupovat a to analogicky k předchozímu příkladu. Jak ale bude vypadat analýza výrazu, pokud jej pozměníme takto: „Existuje množina čísel, o nichž Karel ví, že jsou řešením rovnice 5 mod $x = 1$." V tomto případě nemá Karel vztah k výsledku rovnice (číslu 2 nebo 4), nýbrž ke konstrukci konstruující toto číslo. Typová analýza se rozšíří o výrazy $\exists/(o(o\tau))$ a $Vedet/(o\iota{*_1})_{\tau\omega}$.

$$\lambda w \lambda t\ {}^0\exists \lambda x\, [{}^0Vedet_{wt}\ {}^0Karel\ {}^0[[{}^0 5 \bmod x] = {}^0 1]] \tag{16}$$

Toto však není korektní syntéza, jelikož proměnná x je ve výrazu ^{0}-vázaná. Jak jsme v této kapitole zjistili, vázání Trivializací je silnější než vázání operátorem λ. Kýžené syntézy dosáhneme za použití substituční metody.

$$\lambda w \lambda t\ {}^0\exists \lambda y\, [{}^0Vedet_{wt}\ {}^0Karel\, [{}^0Sub\ [{}^0Tr\, y]\, {}^0x\ {}^0[[{}^0 5 \bmod x] = {}^0 1]]] \tag{17}$$

Pokud Karel ví, že řešením rovnice je číslo 2, pak existuje neprázdná množina čísel, které jsou řešením této rovnice. Pro valuaci $v(2/y)$, to znamená za y substituujeme 2, je výsledkem substituce konstrukce $[[{}^0 5 \bmod {}^0 2] = {}^0 1]$.

V závěru podkapitoly jsou zde uvedeny tři typy kontextu, které se v jazyku TIL vyskytují. Prvním je kontext extenzionální, kdy *hodnota* konstruované funkce je argumentem. Extenzionální výskyt bývá rovněž označován jako výskyt *de re*. Například ve výrazu vyjadřující rovnici

„$\sin(\pi) = 0$“ matematická funkce sinus vystupuje jako funkce, jejíž návratová hodnota se stane argumentem funkce rovnosti. Provedeme-li analýzu datových typů, získáme následující: $0, \pi/\tau$, $\sin/(\tau\tau)$, $=/(o\tau\tau)$. Syntéza výrazu pak vypadá následovně:

$$[[^0\sin\ \pi] = {}^0 0] \tag{18}$$

Druhým typem je kontext intezionální, kdy konstruovaná *funkce* je argumentem. Konstrukce, vyskytující se v intenzionálním kontextu, jsou nazývány také jako *de dicto*. Ku příkladu analyzujme větu: „Sinus je periodická funkce.“ Výraz sinus se zde vyskytuje v intenzionálním kontextu. Datové typy jsou $sinus/(\tau\tau)$ a $periodicka/(o(\tau\tau))$, což je v jazyku TIL množina všech periodických funkcí, tzn. že nám pro argument vrátí pravdivostní hodnotu, zda argument do dané množiny patří. Konstrukci lze vyjádřit syntézou:

$$[^0 Periodicka\ {}^0 Sinus] \tag{19}$$

Zbývá třetí kontext hyperintenzionální, se kterým jsem pracovali například při použití substituční metody v předchozí části této kapitoly. Hyperintenzionální kontext se vyznačuje tím, že samotná *konstrukce* je argumentem funkce. Příklad: „Tom řeší $\sin(x) = 0$“.

Typová analýza: Tom/ι, $resi/(o\iota *_1)_{\tau\omega}$ a význam výrazu „$\sin(x) = 0$“ je konstrukce typu $*_1$. Syntéza:

$$\lambda w \lambda t\, [^0 Resi_{wt}\ {}^0 Tom\ {}^0 [\lambda x\, [[^0 \sin x] = {}^0 0]]] \tag{20}$$

Porozumění těmto základům je stěžejní pro další rozšiřování znalostí v oblasti dynamického diskurzu a řešení anafory. Pro více informací, podrobnou analýzu problematiky a další zajímavé příklady doporučuji monografii *TIL jako procedurální logika* od Marie Duží a Pavla Materny[2].

3.3 Analýza výrazů s anaforickým odkazem

Anafora představuje pragmatický problém, jenž je víc než jen problém logické sémantiky. Tato práce si neklade za cíl vyřešit anaforu, ale popisuje implementaci, která může pomoci při analýze výrazů s anaforickými odkazy. Ve článku Marie Duží [7] je navržen algoritmus, který v této souvislosti pokrývá dvě oblasti. Za prvé dostatečně přesnou analýzou daného diskurzu odhalíme antecedenty vhodného typu[4], které mohou být substituovány za anaforickou referenci[5]. A za druhé existuje-li více čtení dané věty, logická analýza na ně upozorní (termín dvojího čtení bude vysvětlen v této podkapitole). Metoda předpokládá, že antecedent je vždy předem daný, to znamená, že se o něm v dostatečně blízké minulosti mluvilo.

[4] Antecedentem vhodného typu jsou myšleny všechny konstrukce, jež konstruují objekt (příslušného typu) o němž se již hovořilo.

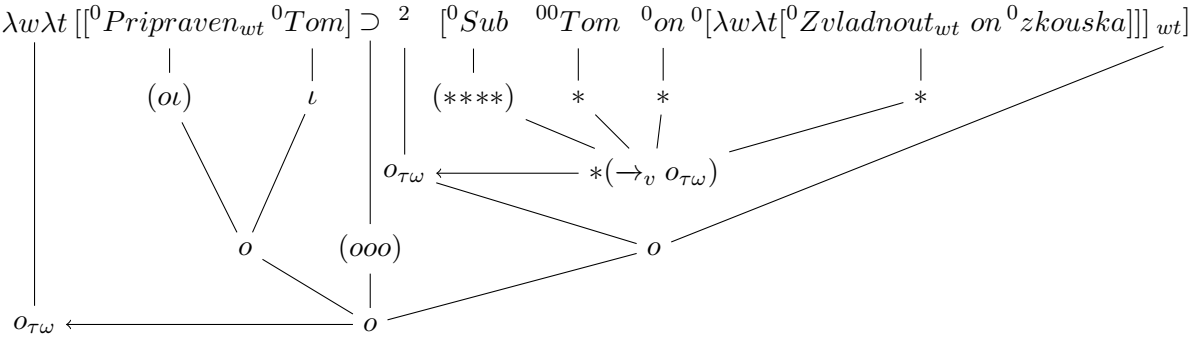
[5] Anaforická reference je výraz zástupného charakteru, proměnná, za níž dosazujeme vhodný antecedent. V přirozené řeči se jedná nejčastěji o zájmeno.

Abychom měli představu, jak vypadá analýza výrazu s anaforickým odkazem, uvádím zde jednoduchý příklad: „Pokud je Tom připraven, pak (*on*) zvládne zkoušku.“ Při analýze výrazu s anaforickým odkazem vždy použijeme substituční metodu.

Analýza datových typů: Tom/ι, $zkouska/\alpha$: objekt je blíže neurčeného typu (jeho přesná specifikace není nezbytná pro analýzu výrazu), $Pripraven/(o\iota)_{\tau\omega}$, $zvladnout/(o\iota\alpha)_{\tau\omega}$, $\supset /(ooo)$, $Sub/(*_1 *_1 *_1 *_1)$ a proměnná $on \rightarrow_v \iota$.

$$\lambda w\lambda t[[^0Pripraven_{wt}\ ^0Tom] \supset\ ^2[^0Sub\ ^{00}Tom\ ^0on\ ^0[\lambda w\lambda t[^0Zvladnout_{wt}\ on\ ^0zkouska]]]_{wt}] \quad (21)$$

V první části věty je zmíněno individuum Tom datového typu ι. Pokud máme správně provedenu analýzu jednotlivých prvků druhé části věty, víme, že anaforická reference *on* konstruuje objekt typu ι. Můžeme za něj tedy dosadit naposledy zmíněný antecedent specifikovaného typu. V této práci se předpokládá, že datový typ anaforických proměnných již byl předem určen. Pro zajímavost proveďme typovou kontrolu, kterou si můžeme prohlédnout na Obrázku 6.



Obrázek 6: Typová kontrola výrazu „Pokud je Tom připraven, pak (*on*) zvládne zkoušku.“

Po vykonání substituce získáme na výstupu konstrukci s úplným významem, neobsahující žádné další anaforické proměnné. Dvojí provedení zde bylo použito tak, že nejprve došlo k aplikaci funkce substituce a poté k provedení konstrukce na výstupu, čímž jsme získali propozici.

$$\lambda w\lambda t[[^0Pripraven_{wt}\ ^0Tom] \supset [^0Zvladnout_{wt}\ \ ^0Tom\ ^0zkouska]] \quad (22)$$

Všímavému čtenáři neuniklo, že jsme rovnou provedli také omezenou beta redukci proměnných w a t, čímž jsme mechanicky zredukovali počet lambda funkcí.

Logika přispívá k vyjasnění nejednoznačností, ku příkladu ve větě „Dítě nemohlo otevřít okno, protože (*ono*) bylo moc těžké.“ Z pohledu formální analýzy výrazu existují dvě možná čtení, vezmeme-li v úvahu, že dítě i okno jsou stejného rodu. Jako zkušení uživatelé českého jazyka jistě správnou verzi uhodneme — zájméno ono neodkazuje k dítěti, nýbrž k objektu okno. Je důležité si uvědomit následující: logika nám nedává návod, jak určit, které čtení je to jediné správné. Poskytuje však nástroje, jak oba výrazy srozumitelně vyjádřit. Empirické zkoumání, co v tomto případě „bylo těžké“, již nespadá do oblasti logiky.

Další nejednoznačností, se kterou se při analýze můžeme setkat, je rozdílné čtení intenzionální (*de dicto*) a extenzionální (*de re*). Uvažujme tyto příklady:

- *de dicto*: Tom si myslí, že (libovolný) drak je nesmrtelný.
- *de re*: Tom si myslí o (tomto) drakovi, že (*on*) je nesmrtelný.

Z pohledu člověka se jedná o jemné rozdíly, které přirozeně vnímáme spíš podvědomě. Pro stroj (počítač) je však stěžejní jednotlivá čtení rozlišovat, aby bylo možné odvozovat vhodné důsledky. To jsou takové, jež nám nemohou do systému či báze znalostí zanést sporné vědomosti (kontradikci).

Mohou nastat situace, kdy lze za anaforický odkaz dosadit více než jeden antecedent daného typu. Vezměme si jako příklad větu: „Chlapec s otcem spatřili draka a chlapec si myslel, že (*on*) je nesmrtelný.“ Pokud bychom vedlejší větu analyzovali samostatně, mohl by vzniknout dojem chlapce uvažujícího o své nesmrtelnosti. Ne vždy je ovšem poslední ve větě se vyskytující antecedent ten správný. Předmětem logiky je tedy najít všechna možná čtení. Rozhodnutí, zda bude některé čtení upřednostněno, je spíš otázkou schopnosti správného usuzování na základě dat z ontologií znalostí.

Algoritmus pro reprezentaci dynamického diskurzu a analýzu výrazů s anaforickými odkazy byl popsán ve článku Marie Duží [7]. Reprezentace dynamického diskurzu sestává z aktualizace seznamu konstrukcí vyjádřených výrazy zmíněnými v diskurzu, abychom v případě potřeby mohli substituovat konstrukci typově vhodné entity za anaforickou proměnnou. Pro jednotlivé typy objektů zmíněných v diskurzu, např. individua (typu ι), vlastnosti individuí (typu $o\iota_{\tau\omega}$), konstrukce (typu $*_n$), apod., vytvoříme seznam konstrukcí těchto objektů. Následně budeme tyto konstrukce dle potřeby dosazovat za anaforické proměnné. Na konstrukce v seznamech se budeme odkazovat pomocí tzv. diskurzních proměnných.

Diskurzní proměnné jsou volné proměnné, které fungují jako paměťové buňky, do kterých jsou postupně ukládány objekty, s nimiž chceme dočasně pracovat (podobně jako proměnné v programovacích jazycích). Tyto hodnoty následně nahradí anaforické proměnné. Nahrazení se provádí pomocí substituční metody.

V průběhu zpracování výrazů si do jednotlivých seznamů ukládáme analyzované konstrukce. Pokud narazíme na větu s anaforickým odkazem, vybíráme ze zásobníku vhodné antecedenty dle typu. Pokud první dosazený objekt není vhodný, zkoušíme dále dosazovat, dokud nedosáhneme požadovaného výsledku — úplného významu věty s anaforickým odkazem.

Nyní si na příkladu ukážeme, jak probíhá tvorba dynamického diskurzu a také substituce diskurzní proměnné za anaforickou proměnnou. V tomto příkladu figuruje agent *Tilman*, jenž implementuje algoritmus pro práci s anaforou a agent *Karel*, který ji při komunikaci rád používá. Níže uvedené výrazy jsou obsahem zpráv, které Tilman obdržel od Karla.

Výraz je nejprve analyzován za použití TIL a případně doplněn pomocí substituční metody. Pro přiřazení konstrukce do diskurzní proměnné používáme značení :=. Diskurzní proměnné

jsou konstrukce typu $*_n$, které ve valuaci konstruují konstrukci entity daného typu. Například diskurzní proměnná *ind* slouží jako odkaz na individua, která byla během komunikace zmíněna. Měli bychom tedy psát $ind/*_n \rightarrow_v *_{n-1}$, resp. $^2ind \rightarrow_v \iota$. Místo toho zapíšeme zkráceně $ind := \iota$. Pokud tedy algoritmus přiřadí do diskurzní proměnné *ind* trivializaci ^0Karel, pak *ind* v-konstruuje ^0Karel, kde $Karel/\iota$.

Každá diskurzní proměnná je tedy uspořádání dříve zmíněných konstrukcí, které konstruují objekty patřičného typu. Seznam je seřazen od naposledy zmíněné ke dříve zmíněným. V příkladu jsou seznamy uzavřeny do ostrých závorek $< \cdots >$. Pokud tedy individua byla zmíněna v pořadí Tom, Alena a naposledy Karel, bude seznam obsahovat objekty v následujícím pořadí: $< {}^0Karel, {}^0Alena, {}^0Tom >$. K jednotlivých objektům pak přistupujeme pomocí indexů tak, že ind_1 odkazuje na konstrukci ^0Karel, ind_2 na objekt ^0Alena, atd. Dle článku je v praxi dostačující udržovat si přehled o posledních pěti antecedentech pro každý typ, což je ovšem pro algoritmus irelevantní. Algoritmus tedy vrací naposledy zmíněnou položku jako první, v případě nejednoznačnosti následně vrací předposlední položku, atd.

Obsah jednotlivých zpráv je očíslován. V každém kroku nejprve provedeme analýzu věty pomocí TIL. Je-li potřeba, nahradíme anaforickou proměnnou vhodným antecedentem pomocí substituční metody. Následně provedeme aktualizaci diskurzních proměnných dle typu. Ve výpise jsou pro stručnost uvedeny pouze diskurzní proměnné relevantní k nové zprávě. Nevypisuje se tedy pokaždé obsah celého diskurzu. V příkladu je použito následujících diskurzních proměnných:

- $ind := \iota$ – indiviua,
- $pred := (o\iota)_{\tau\omega}$ – vlastnosti individuí,
- $prop := o_{\tau\omega}$ – propozice,
- $con := *_n$ – konstrukce,
- $alpha := \alpha$ – objekty blíže neurčeného datového typu[6],
- $rel\_prop := (o\iota o_{\tau\omega})_{\tau\omega}$ – relace v intenzi individua k propozici,
- $rel\_con := (o\iota *_n)_{\tau\omega}$ – relace v intenzi individua ke konstrukci,
- $rel\_alpha := (o\iota\alpha)_{\tau\omega}$ – relace v intenzi individua k blíže neurčenému datovému typu.

Agent Karel posílá zprávy agentu Tilmanovi:

1. „Albert zkoumá závislost mezi energií a hmotou.“
   $\lambda w \lambda t\, [{}^0Zkoumat_{wt}\, {}^0Albert\, {}^0[E = mc^2]]$[7]

[6] Bližší určení datového typu není stěžejní pro další analýzu textu.

[7] Jelikož se v příkladu nesnažíme pracovat s vnitřní strukturou konstrukce „závislost mezi energií a hmotou“ byla pro přehlednost zkrácena do formy vzorce.

- $rel\_con :=< {}^0Zkoumat >$,
- $ind :=< {}^0Albert >$
- $con :=< {}^0[E = mc^2] >$
- $prop :=< \lambda w \lambda t\, [{}^0Zkoumat_{wt}\, {}^0Albert\, {}^0[E = mc^2]] >$.

2. „*(On)* Píše článek.“
   $\lambda w \lambda t\ {}^2[{}^0Sub\ ind_1\ {}^0on\ {}^0[{}^0Psat_{wt}\ on\ {}^0clanek]] \Rightarrow \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Albert\ {}^0clanek]$

   - $rel\_alpha :=< {}^0Psat >$,
   - $alpha :=< {}^0clanek >$,
   - $pred :=< \lambda w \lambda t \lambda on\, [{}^0Psat_{wt}\ on\ {}^0clanek] >$.
   - $prop :=< \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Albert\ {}^0clanek]$,
     $\lambda w \lambda t\, [{}^0Zkoumat_{wt}\, {}^0Albert\, {}^0[E = mc^2]] >$.

3. „Paul také.“[8]
   $\lambda w \lambda t\ {}^2[{}^0Sub\ pred_1\ {}^0take\ {}^0[{}^0take_{wt}\ {}^0Paul]] \Rightarrow$
   $\lambda w \lambda t\, [[\lambda w \lambda t \lambda on\, [{}^0Psat_{wt}\ on\ {}^0clanek]]_{wt}\ {}^0Paul] =_\beta \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Paul\ {}^0clanek]$[9]

   - $ind :=< {}^0Paul, {}^0Albert >$,
   - $prop :=< \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Paul\ {}^0clanek], \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Albert\ {}^0clanek]$,
     $\lambda w \lambda t\, [{}^0Zkoumat_{wt}\, {}^0Albert\, {}^0[E = mc^2]] >$.

4. „Albert věří, že *(on)* *to* vyřeší.“
   $\lambda w \lambda t\, [{}^0Verit_{wt}\, {}^0Albert\ {}^2[{}^0Sub\ ind_1\ {}^0on\, [{}^0Sub\ con_1\ {}^0to\, {}^0[\lambda w \lambda t\, [{}^0Vyresit_{wt}\ on\ to]]]]]$
   $\Rightarrow \lambda w \lambda t\, [{}^0Verit_{wt}\, {}^0Albert\ {}^2[{}^0Sub\ ind_1\ {}^0on\ {}^0[\lambda w \lambda t\, [{}^0Vyresit_{wt}\ on\ {}^0[E = mc^2]]]]]$
   $\Rightarrow \lambda w \lambda t\, [{}^0Verit_{wt}\, {}^0Albert\ \lambda w \lambda t [{}^0Vyresit_{wt}\ {}^0Paul\ {}^0[E = mc^2]]]$

   - $ind :=< {}^0Paul, {}^0Albert >$,
   - $rel\_prop :=< {}^0Verit >$,
   - $rel\_con :=< {}^0Vyresit, {}^0Zkoumat >$,
   - $prop :=< \lambda w \lambda t\, [{}^0Verit_{wt}\, {}^0Albert\ \lambda w \lambda t [{}^0Vyresit_{wt}\ {}^0Paul\ {}^0[E = mc^2]]]$,
     $\lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Paul\ {}^0clanek], \lambda w \lambda t\, [{}^0Psat_{wt}\ {}^0Albert\ {}^0clanek]$,
     $\lambda w \lambda t\, [{}^0Zkoumat_{wt}\, {}^0Albert\, {}^0[E = mc^2]] >$.

Na uvedeném příkladu lze vypozorovat, že poslední analyzovaný výraz je nejednoznačný. Z dosud získaných informací není vůbec zřejmé, zda individuum Paul pracovalo na článku téhož tématu jako individuum Albert a tudíž mělo spadeno na vyřešení stejného problému. Mohlo se

[8] Paul Ehrenfest, teoretický fyzik a blízký přítel Alberta Einsteina.

[9] Ačkoli je zapsán pouze jediný krok *beta*-redukce, zkrátili jsme tím zápis, ve kterém se postupně provedly tři samostatné aplikace pro každou lambda funkci.

klidně stát, že Albert věřil, že zmíněný problém vyřeší on sám. Proč by jej také jinak zkoumal? V takovém případě algoritmus nabídne druhou položku z diskurzní proměnné typu ι. Poslední krok tedy bude vypada následovně (uvádím zde konstrukci již po provedení substituce a jen diskurzní proměnné odlišné od původního řešení):

4'. „Albert věří, že *(on) to* vyřeší.“
$\lambda w\lambda t\,[^0Verit_{wt}\ ^0Albert\ \lambda w\lambda t[^0Vyresit_{wt}\ \ ^0Albert\ \ ^0[E = mc^2]]]$

- $ind :=< {}^0Albert, {}^0Paul >$,
- $prop :=< \lambda w\lambda t\,[^0Verit_{wt}\ ^0Albert\ \lambda w\lambda t[^0Vyresit_{wt}\ \ ^0Albert\ \ ^0[E = mc^2]]]$,
  $\lambda w\lambda t\,[^0Psat_{wt}\ \ ^0Paul\ \ ^0clanek], \lambda w\lambda t\,[^0Psat_{wt}\ \ ^0Albert\ \ ^0clanek]$,
  $\lambda w\lambda t\,[^0Zkoumat_{wt}\ ^0Albert\ ^0[E = mc^2]] >$.

Zmínku si jistě zaslouží také úprava lambda abstrakce, kterou jsme provedli ve druhém kroku. Za použití substituční metody jsme doplnili úplný význam výrazu „Albert píše článek“. Následně jsme provedli lambda abstrakci od konstrukce ^0Albert. Tím jsme získali funkci, která na vstupu očekává argument typu ι, a konstruuje vlastnost individua typu $(o\iota)_{\tau\omega}$. Při abstrakci jsme použili proměnnou *on*.

$$\lambda w\lambda t\lambda on\,[^0Psat_{wt}\ on\ ^0clanek] \tag{23}$$

V této konstrukci je proměnná *on* lambda vázaná. Konstrukci si následně můžeme uložit do diskurzní proměnné. Lambda abstrakce je definována v lambda kalkulu a můžeme ji využít pro vhodné rozšiřování diskurzu. Je třeba mít na paměti, že abstrahovat můžeme jen od konstituentů v uzavřené konstrukci.

Implementace algoritmu pro řešení anafor je hlavní náplní této práce a bude podrobně rozebrána v kapitolách 5 a 6.

4 Komunikace agentů

Během konzultací bylo domluveno, že systém pro zpracování anafory by měl být implementován ve formě samostatné jednotky — agenta, který bude schopen komunikovat s dalšími agenty v rámci multi-agentního systému. V této souvislosti bylo potřeba promyslet dva hlavní aspekty komunikace. Prvním je formát obsahu zpráv, které si agenty mezi sebou vyměňují. Pro tento účel byla zvolena komputační varianta jazyka TIL, která se nazývá TIL-Script. Druhým aspektem je jednotný protokol pro zasílání a příjem zpráv. Na internetu je k dispozici obsáhlá specifikace od společnosti Foundation for Intelligent Physical Agents (FIPA) zabývající se standardizací v oblasti agentů a multi-agentních systémů. Konkrétní specifikace využité v práci jsou popsány v Podkapitole 4.2.

4.1 TIL-script

V předchozích kapitolách jsme probrali některé základní části systému TIL a následně byl popsán návrh algoritmu pro práci s dynamickým diskurzem a řešení anafory. Abychom mohli algoritmus implementovat, potřebujeme definovat jednotný jazyk, se kterým může stroj pracovat. Za účelem uplatnění systému TIL jako součásti software (SW) aplikací byl vyvinut jazyk TIL-Script, který je jeho výpočetní variantou. TIL-Script je deklarativní funkcionální jazyk, který zahrnuje veškerou funkcionalitu systému TIL. Jeho syntaxe se však v některých ohledech liší.

TIL-Script disponuje sadou datových typů, jež vychází z typů definovaných v jazyce TIL. Navíc byly přidány datové typy `Int` a `String`, a zároveň byl typ τ rozdělen na dva datové typy `Time` a `Real`. Dále namísto specifikace n-tého řádu konstrukce se v TIL-Scriptu používá obecné označení konstrukce `*`. Výčet všech datových typů je k dispozici v Tabulce 2.

TIL-Script	TIL	Popis
`Bool`	o	pravdivostní hodnoty $\{P, N\}$
`Indiv`	ι	individua
`World`	ω	možné světy
`Time`	τ	možné časy
`Real`	τ	reálná čísla
`Int`		celá čísla
`String`		řetězec znaků
`*`	$*_n$	konstrukce n-tého řádu

Tabulka 2: TIL-Script: Datové typy

Dle původní specifikace TIL-Scriptu se pracovalo pouze se znaky ASCII tabulky. Z toho důvodu v něm nebylo možno použít horní či dolní indexy a některé speciální znaky (např. operátor lambda, znaky kvantifikátorů, apod.). Tyto jsou v TIL-Scriptu zastoupené symboly z ASCII kódu. Tabulka 3 zobrazuje seznam symbolů, se kterými jsem se při práci setkala. V prvním

znak	TIL-Script	TIL	Popis
\	`\x C`	$\lambda x\, C$	operátor lambda
:	`\x:Indiv`	$x \rightarrow_v \iota$	definice typu objektu v-konstruovaného λ-vázanou proměnnou
,	`\x, y`	$\lambda xy\, C$	čárka odděluje proměnné ve víceargumentové lambda funkci
’	`'C`	${}^0C$	apostrof značí trivializaci
^	`^2C`	${}^2C$	symbol uvozující n-té provedení (v příkladu dvojí)

Tabulka 3: TIL-Script: Speciální znaky

sloupci je uveden znak a druhý sloupec pak demonstruje jeho použití v notaci TIL-Scriptu. Ekvivalentní zápis v TIL je znázorněn ve třetím sloupci, setkali jsme se s ním v předchozích kapitolách. Poslední sloupec popisuje funkci symbolu. V TIL se obdoba znaku ’`:`’ nevyužívá přímo v konstrukcích, ale při analýze typů se uvádí, jakého typu je objekt v-konstruovaný λ-vázanou proměnnou.

Kvantifikátory a další zavedené funkce jsou v TIL-Scriptu nahrazeny klíčovými slovy, například `Exist`, `Every`, `Improper`, `Sub` atd.

Později vyvstala potřeba analyzovat pomocí gramatiky také konstrukce obsahující česká znaménka interpunkce. Aktuální verze tedy podporuje také písmena české abecedy v kódování UTF-8 a je uložena mezi přílohami v souboru `TIL-Script grammar ebnf v5.1.txt`.

Všimněme si nyní, jak je definován tvar Uzávěru s operátorem lambda. Dle gramatiky je možno definovat proměnnou bez uvedení typu. Řešení anafory však vyžaduje, aby byla typová analýza provedena úplně, to znamená, že každá procedura včetně jejích podprocedur má analýzou přiřazený datový typ. To se týká také lambda proměnných. Bez této analýzy konstrukce neprojde typovou kontrolou a neměla by tedy ani přijít na vstupu pro algoritmus řešení anafor.

Rovněž v TIL-Scriptu je nutno rozlišovat mezi více vnořenými lambda funkcemi a víceargumentovými funkcemi, jak již bylo ukázáno pro TIL u konstrukce 6. Jen pro úplnost je zde uveden stejný příklad v notaci TIL-Scriptu, ovšem bez provedení beta redukce. Na druhém řádku je uvedena také varianta se dvěmi vnořenými lambda funkcemi. Každá konstrukce je v TIL-Scriptu ukončena tečkou.

```
[[\x:Int, y:Int [’- x y]] ’8 ’5].
[[[\x:Int [\ y:Int [’- x y]]] ’8] ’5].
```

Výpis 1: Dvouargumentová lambda funkce a dvě vnořené lambda funkce v TIL-Scriptu

Jazyk TIL-Script slouží také pro zápis dat do báze znalostí, kterou mohou agenty využívat ku příkladu pro logické usuzování. Nestačí-li agentu pro vyhodnocení úplného významu věty vlastní znalostní báze, může komunikovat s dalšími agenty a prostřednictvím vhodného dotazování získat chybějící znalosti. Práce s bází znalostí a související dotazování není předmětem této práce.

Soubory obsahující TIL-Script mají příponu `.tils`. Mezi přílohami se nachází také následující soubor: `example of TIL-Script message.tils` obsahující tři výrazy přirozeného jazyka analyzované v jazyce TIL-Script.

Nyní již víme, že agenty komunikují v jazyce TIL-Script. Při konzultacích bylo dohodnuto, že zprávy obsahující TIL-Script budou ve formátu Extensible Markup Language (XML). V principu by XML formát měl obsahovat tytéž informace, jaké jsou vyjádřeny v souboru s příponou `.tils`. Struktura XML aktuálně není nikde definována, v práci tedy vycházím z informací získaných během konzultací a poskytnutého příkladu XML souboru. Tento soubor lze nalézt mezi přílohami pod názvem `example of TIL-Script message.xml` a obsahuje analýzu stejných tří výrazů, jako soubor `.tils`. Jedná se o jediná ukázková vstupní data, která jsem při implementaci měla k dispozici.

Jelikož XML je strukturováno do stromové struktury, popíšeme si jej jako strom. Ten je uveden kořenem `source`, z něhož vedou tři základní větve: `entities`, `variables` a `constructions`. První větev obsahuje výčet všech entit, jež se v souboru s analýzou vyskytují. Jednotlivé listy se nazývají `entity` a jsou jednoznačně identifikovány hodnotami atributů `name` a `type`. Výpis entit znázorněný ve Výpise 2 lze srovnat se zápisem $Tilman/\iota$, Pi/τ v TIL nebo `Tilman/Indiv.` a `Pi/Real.` v TIL-Scriptu.

```
<entities>
  <entity name="Tilman" type="Indiv"/>
  <entity name="Pi" type="Real"/>
</entities>
```

Výpis 2: Výpis entit v TIL-Script XML

Když jsme při analýze v Kapitole 2 prováděli výčet typů, uváděli jsme také proměnné. Volné proměnné jsou v XML specifikovány ve druhé větvi `variables`. Tato větev byla ve vstupních datech, narozdíl od zbylých dvou větví, prázdná. Struktura jednotlivých proměnných, jak jsem ji odvodila, je znázorněna ve Výpise 3. List `variable` vyjadřuje, že proměnná *on* v-konstruuje objekt typu ι. V jazyce TIL-Scriptu bychom napsali `on -> Indiv.`.

```
<variables>
  <variable name="on" type="Indiv"/>
</variables>
```

Výpis 3: Výpis proměnných v TIL-Script XML

Je potřeba zdůraznit, že sekce `variables` obsahuje pouze volné proměnné. Pokud je tedy v konstrukci λ-vázaná nebo ^{0}-vázaná proměnná, pak není v této sekci uvedena.

Poslední větví je ta s názvem `constructions`. V ní je uvedena analýza jednotlivých konstrukcí tak, že každý výraz přirozeného jazyka je zde chápán jako samostatná analyzovaná konstrukce s výčtem atributů a svou strukturou charakterizuje skládání konstrukce a jejích podkonstrukcí do stromu. Struktura stromu je srovnatelná s abstraktním syntaktickým stromem vyobrazeným na Obrázku 5.

Sekce `constructions` je znázorněna ve Výpise 4 a nyní si ji postupně projdeme. Pod uzlem `constructions` jsou větve. Každá větev zahrnuje jeden výraz analyzovaný formou TIL-Script procedury a začíná uzlem charakterizujícím analýzu konstrukce jako celku.

```
<constructions>
  <construction ID="#1" constructionType="composition" occurrence="Intensional"
      construction="[[\x:Int ['- x '5]] '8]" type="Int">
    <construction ID="#1#1" constructionType="closure" occurrence="Extensional"
        construction="[\x:Int ['- x '5]]" type="(Int Int)">
      <construction ID="#1#1#1" constructionType="composition" occurrence="Intensional"
          construction="['- x '5]" type="Int">
        <construction ID="#1#1#1#1" constructionType="trivialisation" occurrence="
            Extensional" construction="'-" type="(Int Int Int)">
        </construction>
        <construction ID="#1#1#1#2" constructionType="variable" occurrence="Intensional
            " construction="x" type="Int">
        </construction>
        <construction ID="#1#1#1#3" constructionType="trivialisation" occurrence="
            Intensional" construction="'5" type="Int">
        </construction>
      </construction>
    </construction>
    <construction ID="#1#2" constructionType="trivialisation" occurrence="Intensional"
        construction="'8" type="Int">
    </construction>
  </construction>
</constructions>
```

Výpis 4: Výpis konstrukcí v TIL-Script XML

Konstrukce je specifikována hodnotami jednotlivých atributů. Atribut `ID` definuje strukturu, tzn. ve které části stromu se daná konstrukce nachází. Atribut shledávám redundantní vzhledem ke struktuře konstrukce vyjádřené samotným stromem. Další atribut `constructionType` nabývá vždy jedné z hodnot vyjádřených v gramatice TIL-Scriptu na pravé straně neterminálu *construction*. Ve Výpise 4 jsou kromě dvojího provedení (`n-execution`) zachyceny všechny možné hodnoty, jakých může atribut nabývat. Nutno zdůraznit, že v případě dvojího provedení by `constructionType` nabýval hodnoty `2-execution`. Samotné provedení (^1C) se v TIL-Scriptu explicitně nepoužívá.

Následujícím atributem `occurrence` je vyjádřeno, v jakém kontextu se daná konstrukce vyskytuje. Jak již víme, v TIL pracujeme se třemi typy kontextu: *extenzionálním* (hodnota argumentu `Extensional`), *intenzionálním* (hodnota `Intensional`) a *hyperintenzionálním* (hodnota `Hyperintensional`). V atributu `construction`, možná trochu nešťastně pojmenovaném shodně s názvem uzlu, je vyjádřena konstrukce v jazyce TIL-Script. Posledním atributem je `type`, který specifikuje datový typ konstruovaný danou konstrukcí.

Potomky uzlu jsou podkonstrukce, z nichž se konstrukce skládá. Jedná-li se o konstrukci Kompozice, pak má uzel dva a více potomků, což je vždy konstrukce funkce a jejích argumentů.

Pracujeme-li s konstrukcí uzávěru, pak je zde jediný potomek vyjadřující lambda abstrakci. Trivializace konstruuje atomickou konstrukci, např. trivializace individua `'Tilman`, nebo molekulární konstrukci (Kompozice, Uzávěru, apod.). Proměnná je vždy atomická konstrukce.

4.2 ACL zprávy

V multi-agentním systému se většinou vyskytují agenty, které mají často rozdílnou funkci a implementaci. Za účelem dosažení schopnosti agenta komunikovat v multi-agentním systému byla vyvinuta rozsáhlá specifikace společností FIPA, jež v současnosti spadá pod organizaci IEEE Computer Society [8]. V této podkapitole jsou popsány části specifikace využité při implementaci. Jelikož se jedná o velmi rozsáhlý repozitář popisující architekturu, koncept komunikace, formát zpráv a mnoho dalšího, jsou pro přehlednost v textu odkazovány identifikátory souvisejících dokumentů.

FIPA specifikace abstraktní architektury (dokument *SC00001L*) popisuje, jak se mohou dva agenty lokalizovat (vzájemná registrace) a následně spolu komunikovat vyměňováním zpráv. Výměna pak reprezentuje řečový akt, který je kódovaný v jazyce Agent Communication Language (ACL). Konkrétní architektura by správně měla zahrnovat všechny části abstrakce, pro účely této práce však byly některé části upraveny nebo zcela vynechány. Odlišnosti od specifikace jsou popsány v Kapitole 5. Důležité mechanizmy jsou registrace agentů a přenos zpráv mezi nimi. Takový ideální agent by měl být schopen jednat autonomně a projevit sémantickou interoperabilitu[10] na základě zamýšleného záměru.

Registraci agentů zajišťuje *Agent Directory Services*, v níž se uchovává popis jednotlivých záznamů o agentech. Záznam nezbytně obsahuje globálně unikátní jméno agenta a popis, jakým způsobem je možné se s agentem spojit a komunikovat, tedy typ přenosu a adresa specifická pro tento typ. Pokud se chce agent zaregistrovat, může ku příkladu začít poslouchat na specifickém portu, čímž začne být dosažitelný pro ostatní agenty. Pro nalezení agenta se může použít zmíněná *Agent Directory Services*. Z této služby získá agent záznam jiného agenta, se kterým může komunikovat. Další údaje blíže popisující agent je také možno uvést, například s jakými ontologiemi umí pracovat. Uživatelem definované parametry začínají prefixem „X-“.

Další částí architektury je *Service Directory Services*, jež poskytuje způsob, jak může agent nalézt služby. Může se jednat ku příkladu o server poskytující ontologickou bázi znalostí. Princip je obdobný jako u *Agent Directory Services*.

Poslední důležitou součástí architektury, kterou zde zmiňuji, je *Agent Messages*. Tato část se skládá z popisu struktury, reprezentace a přenosu zprávy. Struktura je psaná v ACL a obsah zprávy je vyjádřen v *content-language*, což může být například Prolog nebo v našem případě TIL-Script. Obsah může být navázaný na nějakou ontologii. Zpráva obsahuje také odesílatele a žádného, jednoho nebo více příjemců. Pokud není příjemce dán, probíhá komunikace na bázi

[10] schopnost systémů vzájemně si poskytovat služby a efektivně spolupracovat

broadcastingu[11]. Zprávy mohou rekurzivně obsahovat další zprávy. ACL zpráva může být zakódována do tzv. *payloadu*, který je následně vložen do transportní zprávy. Ta obsahuje navíc tzv. obálku, v níž jsou uvedeny již konkrétní adresy odesílatele a příjemce pro daný přenos. Transportní zpráva může mít formu Hypertext Transfer Protocol (HTTP) požadavku, TCP zprávy, apod.

Je doporučeno identifikovat agent jménem, které je odlišné od údajů popisujících přenos. Můžeme to demonstrovat na příkladu. Pokud agent B provádí usuzování o agentech, se kterými komunikuje, může použít jméno agenta A namísto použití třeba jeho IP adresy. V takovém případě bude schopen zachovat souvislost usuzování i v případě změny způsobu přenosu.

FIPA specifikace struktury ACL zprávy popisuje jednotlivé parametry vyskytující se v ACL zprávě. Jaké parametry budou ve zprávě obsaženy je dáno v závislosti na situaci. Výčet všech parametrů uvádí dokument *SC00061G*. V rámci této práce se používají především parametry z Tabulky 4.

Parametr	Kategorie
`performative`	typ řečového aktu
`sender`	účastník komunikace
`receiver`	účastník komunikace
`reply-to`	účastník komunikace
`content`	obsah zprávy
`language`	popis zprávy
`conversation-id`	řízení konverzace

Tabulka 4: Výčet některých parametrů ze struktury ACL zprávy

Performativ (`performative`) je jediný povinný parametr, očekává se však, že většina zpráv obsahuje také odesílatele (`sender`), příjemce (`receiver`) a obsah zprávy (`content`). Odesílatel je vždy pouze jeden a hodnota parametru je dána názvem agenta, který zprávu odesílá. Hodnota parametru odesílatel může být název jednoho agenta nebo neprázdná množina jmen. Druhá možnost se používá při způsobu komunikace známé jako *multicast*, což lze přirovnat k hovoru ve skupině. Například agent A uskuteční tzv. `inform` akt na množině příjemců, čímž je chce přesvědčit, aby věřili obsahu jeho zprávy.

Parametr `reply-to` zajišťuje, že následující zpráva konverzace bude přesměrována na agent specifikovaný v tomto parametru. Není-li tento parametr specifikován, používá se pro určení příjemce odpovědi hodnota v parametru `sender`. Obsah zprávy (`content`) musí být uveden v naprosté většině případů. Parametr `language` specifikuje, v jakém jazyce je vyjádřen obsah zprávy. Poslední zde uvedený parametr je `conversation-id`, jehož hodnota je výraz (identifikátor konverzace) sloužící pro identifikaci probíhající posloupnosti řečových aktů tvořících konver-

[11]Broadcasting je způsob síťové komunikace, při níž odesílatel nekontaktuje konkrétního příjemce, ale posílá data najednou všem příjemcům v určitém síťovém rozsahu.

zaci. Uvedením tohoto parametru může agent snadno rozlišit individuální konverzace s různými agenty nebo skupinami agentů. Zpráva může obsahovat také parametry definované uživatelem. Takové parametry je třeba uvozovat prefixem „X-“ (např. `X-speech`).

Ve specifikaci FIPA knihovny řečových aktů (*SC00037J*) je uveden seznam performativů, které můžeme při komunikaci použít pro určení účelu zprávy. Dle tohoto dokumentu je doporučeno, aby agenty implementovaly alespoň typ `not-understood`. Tabulka 5 zahrnuje typy řečových aktů použitých v této práci. Úplný výčet je k dispozici ve zmíněné specifikaci.

Akt	Popis
`agree`	souhlas s vykonáním nějaké akce
`confirm`	odesílatel potvrzuje příjemci pravdivost dané propozice
`failure`	vyjadřuje, že snaha vykonat nějakou akci se nezdařila
`inform`	odesílatel informuje příjemce o pravdivosti uvedené propozice
`not-understood`	vyjadřuje neporozumění zprávě, kterou agent obdržel
`query-ref`	dotaz na objekt odkazovaný v obsahu zprávy
`refuse`	odmítnutí vykonat nějakou akci s vysvětlením důvodu

Tabulka 5: Výčet některých performativů z FIPA knihovny řečových aktů

Variantou aktu `query-ref` je akt vyjádřený performativem `query-if`, prostřednictvím kterého se nedotazujeme na objekt, ale zda je propozice uvedená v obsahu zprávy pravdivá či nikoli. Jako příklad komunikačního aktu může sloužit specifikace FIPA protokolu pro interaktivní dotazování (*SC00027H*). V tomto dokumentu figuruje iniciátor komunikace a účastník. V prvním kroku iniciátor zasílá dotaz účastníkovi. Je-li dohodnuto, že agent zasílá předem notifikaci o svém rozhodnutí, zda akci provede, pak účastník odpovídá `refuse` nebo `agree`. Souhlasil-li účastník, pak může začít s provedením kroků potřebných pro zjištění odpovědi na dotaz iniciátora. Závěrečná odpověď vyjadřuje, zda jeho kroky byly neúspěšné (`failure`) nebo naopak úspěšné. V případě úspěchu zasílá účastník `inform` zprávu. Pokud byl dotaz typu `query-if`, pak odpověď obsahuje informaci, zda je daná propozice pravdivá. Při `query-ref` je v odpovědi specifikován objekt odkazovaný v dotazu iniciátora.

Poslední specifikace, kterou zde zmíním, je FIPA reprezentace ACL zprávy v podobě řetězce (*SC00070I*), která jasně definuje syntaxi a je napsaná ve standardní rozšířené Backus-Naurově formě (EBNF), která navíc oproti základní verzi umí pracovat i s regulárními operátory. Příklad ACL zprávy vytvořené pomocí této gramatiky je zobrazen ve Výpise 5. Agent je zde označen klíčovým slovem `agent-identifier` a je možné mu nastavit hodnoty několika parametrů. Do povinného parametru `name` uvádíme jméno agenta. Parametr `addresses` je definován jako sekvence Uniform Resource Locator (URL) adres. Dalším parametrem je `resolvers`, obsahující sekvenci agentů, které poskytují službu *name resolution*. Jedná se o vyhledávací funkci, jež je součástí *Agent Management System* (viz specifikace *SC00023J*). Agent může tyto agenty kon-

taktovat za účelem zjištění transportní adresy či adres jiného agenta. Agentu lze nastavit i další uživatelem definované parametry. Názvy těchto parametrů začínají prefixem „X-“.

```
(inform
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "weather (today, raining)"
  :language Prolog)
```

Výpis 5: Ukázka struktury ACL zprávy pro řečový akt `inform`

5 Specifikace požadavků a návrh algoritmu

Předchozí Kapitola 4 se již částečně dotýkala specifikace požadavků a to konkrétně ve smyslu, jakým způsobem jsou zprávy reprezentovány a jak jsou posílány v rámci multi-agentního systému. V této kapitole uvádím, na jakých vlastnostech systému jsem se dohodla s vedoucí práce. Zmíním také odlišnosti od dříve popsaných specifikací jazyka ACL.

Druhá část této kapitoly navrhuje již konkrétní implementaci algoritmu pro zpracování konstrukcí s anaforickými odkazy a rovněž řešení komunikace agenta v multi-agentním systému.

5.1 Specifikace

Zadání práce popisuje vstup jako databázi konstrukcí jazyka TIL-Script. Na konzultacích se dohodlo, že konkrétně se bude jednat o konstrukce jazyka TIL-Script ve formátu XML po provedení typové analýzy. Nemělo by se tedy stát, že na vstupu přijde XML, které neprošlo typovou kontrolou. Vstupy by měly dodávat jiné agenty komunikující v rámci multi-agentního systému.

Očekává se, že na výstupu program vydá systém konstrukcí, ve kterých jsou dynamicky doplněny anaforické odkazy na dříve zmíněné objekty. Také v tomto případě bude použit stejný formát jako na vstupu. Doplnění úplného významu věty bude provedeno včetně provedení substituční metody. Na výstupu tedy bude konstrukce s doplněnou entitou a nikoli konstrukce s funkcí substituce. Pro jeden vstupní soubor vznikne jeden výstupní soubor bez ohledu na počet konstrukcí analyzovaných v rámci jednoho souboru.

Agent bude očekávat vstupní data prostřednictvím internetového protokolu *Transmission Control Protocol* (TCP). Data dorazí ve formátu ACL zprávy obsahující TIL-Script XML. Výrazy budou v anglickém jazyce.

Původní požadavek zněl, aby byl systém pro řešení anafory řešen formou zásuvného modulu do implementace agenta dodaného vedoucí práce. Paralelně s touto prací totiž vznikají další systémy, jež implementují jiné moduly, které by pak agent mohl používat. Později se ukázalo, že toto řešení je komplikované, jelikož konkrétní implementace nevyhovovala dalším požadavkům na aplikaci (např. způsob komunikace prostřednictvím UDP, zprávy ve formátu prostého textu, chybějící dokumentace, nespecifikované rozhraní pro zásuvné moduly, apod.). Z tohoto důvodu bylo rozhodnuto, že implementace systému pro anafory sice bude formou zásuvného modulu, ale dojde také ke specifikaci rozhraní. Pro zajímavost jsou mezi přílohami také zdrojové kódy původního agenta.

Systém pro řešení anafory by měl být schopen tvořit „zásobníky“ konstrukcí dle typů objektů, které konstruují. Při každé příchozí zprávě se aktualizují relevantní zásobníky. Naposledy aktualizované objekty jsou v zásobnících nejvýš. Princip tvorby zásobníku je popsán v příkladu na konci Kapitoly 3. Obsahuje-li příchozí zpráva objekt, který se už v zásobníku nachází, je tento smazán a přesunut na vrchol zásobníku. Velikost zásobníku byla stanovena na 10 referencí, ale tuto hodnotu bude možné změnit konfigurací. Postup substituce bude možné si prohlédnout v uživatelském rozhraní.

Během konzultací bylo dohodnuto, že některé části specifikace FIPA budou upraveny nebo zcela vynechány. Jedná se o následující odlišnosti:

- agenty budou identifikovány jménem ve tvaru `<IP Adresa>:<Port>`,
- možnost nastavit adresu agenta ve tvaru IP adresy a portu,
- ACL zpráva bude kromě povinného performativu vždy obsahovat parametry: `sender`, `receiver`, neprázdný `content` a `language`,
- algoritmus neimplementuje funkcionalitu `Resolvers` a `Attributes` z FIPA specifikace,
- agenty si registují jiné agenty tím, že od nich přijmou zprávu.

Agent bude dále schopný odpovědět na specifický dotaz `query-ref` tak, že vrátí jména všech agentů, které zná. Seznam známých agentů bude tvořen průběžně během komunikace. To znamená, že když agent přijme zprávu, uloží si jméno odesílatele do databáze.

V závěru podkapitoly uvádím seznam požadavků na program, které byly probrány v předchozích odstavcích.

1. vstup: konstrukce v jazyce TIL-Script ve formátu XML po provedení typové analýzy,
2. výstup: konstrukce v jazyce TIL-Script ve formátu XML, ve kterém byly anaforické odkazy nahrazeny vhodnými, dříve zmíněnými objekty,
3. schopnost aplikace přijmout zprávy na protokolu TCP,
4. výrazy jsou v angličtině,
5. systém pro řešení anafory formou zásuvného modulu,
6. specifikace rozhraní zásuvného modulu,
7. aktualizace diskurzních proměnných s každou příchozí zprávou,
8. velikost „zásobníku“ stanovena na 10 referencí s možností úpravy v konfiguraci,
9. zobrazení postupu substituce v uživatelském rozhraní,
10. možnost nastavit adresu agenta také ve tvaru `<IP Adresa>:<Port>`,
11. schopnost vrátit seznam jmen agentů, které agent zná.

5.2 Návrh algoritmu

Systém pro řešení anafory očekává na vstupu analýzu konstrukcí v TIL-Scriptu ve formátu XML. Tato data by měl poskytnout agent, který byl vyvíjen v rámci jiné diplomové práce paralelně s touto prací. V době implementace však nebyla vstupní data k dispozici. Bylo tedy potřeba vytvořit si vstupní data ručně, aby bylo možné algoritmus otestovat.

Pokud má být algoritmus řešící anafory vytvořen formou zásuvného modulu, je nutno navrhnout vhodné rozhraní, které bude algoritmus implementovat. Společná vlastnost všech zásuvných modulů bude ta, že na vstupu obdrží TIL-Script XML, provedou s ním nějaké specifické operace a na výstupu vrátí opět TIL-Script XML. Specifické operace jednotlivých zásuvných modulů mohou být např. doplnění anafory, kontrola datových typů, analýza kontextů, apod. Předpokládá se, že bude potřeba modul vhodně konfigurovat, měli bychom tedy mít možnost nastavit jeho specifické parametry. Konkrétní parametry se však mohou lišit modul od modulu. Proto by neměl být jejich počet ani datový typ přesně specifikovaný. Návrh rozhraní pro zásuvné moduly může vypada například tak, jak je vyobrazeno na Výpise 6.

```
interface IPlugin
{
    TILScript Execute(TILScript xml);
    void SetParameters(string[] parameters);
}
```

Výpis 6: Návrh rozhraní pro zásuvné moduly

Jakmile získáme vstupní data v požadovaném formátu, můžeme s nimi začít pracovat. Modul anafory musí umět ze vstupních dat budovat databázi konstrukcí, které byly zmíněné v diskurzu. Konstrukce jsou dány unikátní kombinací tvaru konstrukce (atribut `construction`) a typu, který je z ní konstruovaný (atribut `type`). Tvar konstrukce budeme v následujcím textu nazývat hodnotou `value` pro jeho odlišení od názvu uzlu. Diskurzní proměnné jsou také unikátní svou hodnotou a typem v daném diskurzu. Za účelem správného řazení tak, aby naposledy zmíněné konstrukce byly k dispozici první, opatříme každou diskurzní proměnnou časovým razítkem. Při aktualizaci:

a) uložíme do diskurzu nové proměnné s aktuálním časovým razítkem,

b) existuje-li již proměnná v diskurzu, pak aktualizujeme její časové razítko.

Agent by měl být schopen účastnit se více konverzací současně a nemusí jít nutně jen o konverzaci ve dvojici, ale i v rámci skupiny. Kdyby měl agent pro veškeré konverzace sdílený diskurz, je velmi pravděpodobné, že při dosazení za anaforickou proměnnou někdy dosadí i konstrukce, o kterých se v dané konverzaci nehovořilo. Vhodně oddělit data z různých konverzací je možné například identifikací diskurzu podle jména či jmen agentů, se kterými agent komunikuje. Ještě

vhodnější je mít speciální identifikátor konverzace. Pokud by skupina ku příkladu prováděla usuzování o agentech, kteří v ní figurují, pak i v případě, že některý z účastníků konverzaci opustí, je zbytek skupiny pořád schopný udržet souvislost usuzování. Této výhody můžeme dosáhnout využitím parametru `conversation-id`, do kterého agent zahajující konverzaci uloží globálně unikátní hodnotu. Tento identifikátor se pak používá ve všech zprávách v konverzaci.

Uvažujme případ, kdy agent A hovoří zároveň s agenty B a C. Při komunikaci s agentem B si předávají citlivé údaje, které by měly zůstat skryty agentu C. Použití unikátních identifikátorů konverzace umožní oddělit jednotlivé konverzace. Zároveň může být `conversation-id` použito také pro usuzování o historických záznamech dané konverzace. Pokud by nebylo ve zprávě zahajující komunikaci `conversation-id` uvedeno, je to možné řešit vytvořením samostatného diskurzu, který bychom mohli nazvat *veřejný*, a u něhož by v případě úniku nějaké informace nehrozila kompromitace některého z agentů.

Je také důležité zamyslet se nad otázkou, co vše do diskurzu ukládat. Při aktualizaci diskurzu provedeme rozklad procedury na konstituenty. Jistě dává smysl ukládat atomické konstrukce, např. `'Karel`, `'Student` (konstituenty) a také proceduru jako celek. Obecně tedy platí, že do diskurzu budeme ukládat všechny uzavřené konstrukce. Pro názornost se můžeme podívat zpět na Výpis 4. Z konstrukcí, uvedených ve výpise, neuložíme do diskurzu pouze otevřenou konstrukci `['- x '5]` a konstrukci proměnné `x`.

Dále je na zvážení, zda jít i do hyperintezionálního kontextu. Mějme konstrukci `'['Lastdec 'Pi]`, kterou jsme použili ve větě: „Tilman seeks last decimal of Pi.“ V tomto případě by se dalo odkázat do hyperintezionálního kontextu třeba větou: „This constant has infinite number of decimal digits.“ Proto také uzavřené podkonstrukce z hyperintenzionálního kontextu budeme do aktualizace zahrnovat.

Je zde také možnost ukládat do diskurzu konstrukce, ve kterých jsme abstrahovali od nějaké konkrétní hodnoty. Jako příklad uvažujme větu o Tilmanovi v předchozím odstavci. V této větě můžeme abstrahovat od individua *Tilman*, čímž získáme proceduru konstruující vlastnost individua „seeking last decimal of π.“ Abstrahovat můžeme od různých typů konstrukcí, v algoritmu anafory bude možnost abstrakce od individua s možností dalšího rozšíření.

V souvislosti s dynamickým diskurzem zbývá otázka, kdy provádět aktualizaci diskurzu. Pro věty s úplným významem je to jasné. Aktualizaci provádíme, když takovou větu obdržíme. Pokud však obsahuje anaforické proměnné, dáva smysl provést aktualizaci před dosazením za anaforickou proměnnou a samozřejmě také po dosazení. V prvním případě bude množina aktualizovaných diskurzních proměnných omezena neúplným významem věty, pro příklad ve větě „The boy and his daddy saw a dragon, and the boy thought that he was immortal.“ Kdybychom neprovedli aktualizaci diskurzu před dosazováním anafory, neobsahoval by diskurzní proměnnou *dragon*.

Na Obrázku 7 je ke shlédnutí diagram řídícího toku pro aktualizaci dynamického diskurzu. Algoritmus začíná kontrolou, zda je XML na vstupu validní. Můžeme si vytvořit soubor XML Schema Definition (XSD), kterým budeme validitu kontrolovat. Následně provedeme kontrolu

vstupních parametrů pro algoritmus. Pokud byl dodán parametr `conversation-id`, použijeme jeho hodnotu k určení aktuálního diskurzu. V opačném případě použijeme *veřejný* diskurz. Pokud obě kontroly projdou, lze pokračovat k iteraci přes seznam konstrukcí. Iterace je prováděna rekurzivně, to znamená, že pro každou konstrukci procházíme také její potomky až k listům — atomickým konstrukcím. Pro každou konstrukci, která splňuje definované podmínky, provedeme aktualizaci diskurzu. Vždy, když během konverzace dojde k aktualizaci diskurzu, je třeba začít s hledáním naposledy použitých diskurzních proměnných znova od nejnovějších záznamů.

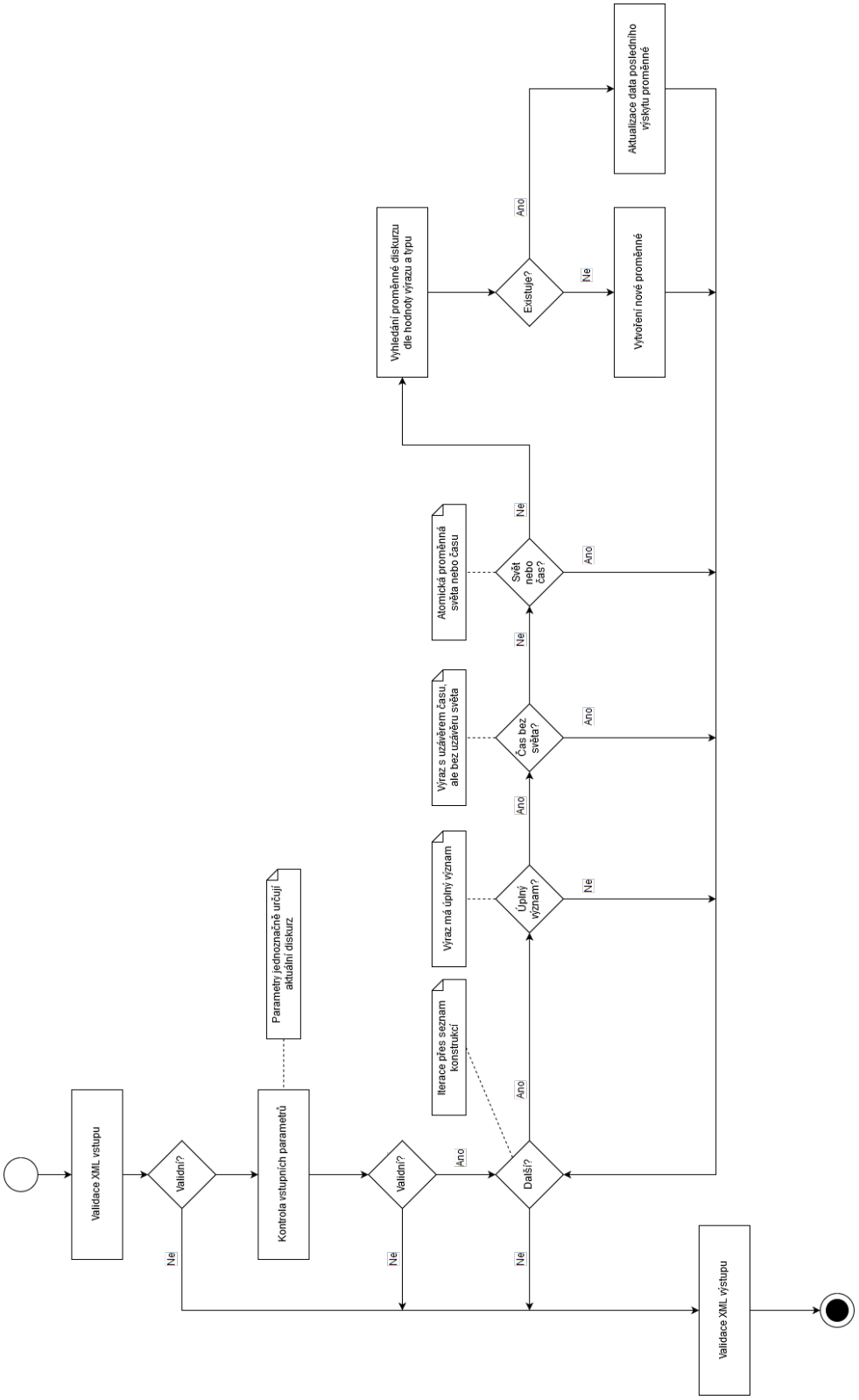
Další klíčovou částí algoritmu je substituce za anaforické proměnné. Konstrukce obsahuje anaforickou proměnnou, je-li daná konstrukce otevřená. Pokud takovou proměnnou identifikujeme, můžeme postupovat dle diagramu řídícího toku znázorněném na Obrázku 8. Prvním krokem je opět kontrola validity vstupního XML. Následně vezmene jednu konstrukci ze vstupního XML po druhé a pro každou provedeme následující kroky. Pokud konstrukce obsahuje anaforu, provedeme kontrolu vstupních parametrů, jež jednoznačně určují aktuální diskurz. Pak ze struktury konstukce získáme seznam všech anaforických referencí. Dalším krokem je provádění iterace přes jednotlivé reference.

Každá anaforická proměnná v konstrukci by měla mít související záznam v sekci `variables` XML vstupu se stejným typem. Pro danou konstrukci získáme naposledy použitou diskurzní proměnnou, kterou jsme za anaforu dosazovali. Pokud žádná taková neexistuje, to znamená, že jsme ještě nikdy do celé konstrukce nezkoušeli za danou anaforu nic dosazovat, pak vezmeme naposledy aktualizovanou diskurzní proměnnou daného typu. V případě, že jsme do celé konstrukce dříve již zkoušeli něco dosazovat, vybereme další nejaktuálnější proměnnou v pořadí. Neexistuje-li žádná taková, pak pokračujeme k další anaforické proměnné, případně je algoritmus ukončen. Ovšem pokud existuje, pak definujeme substituci obsahu diskurzní proměnné za anaforickou proměnnou a následně ji provedeme.

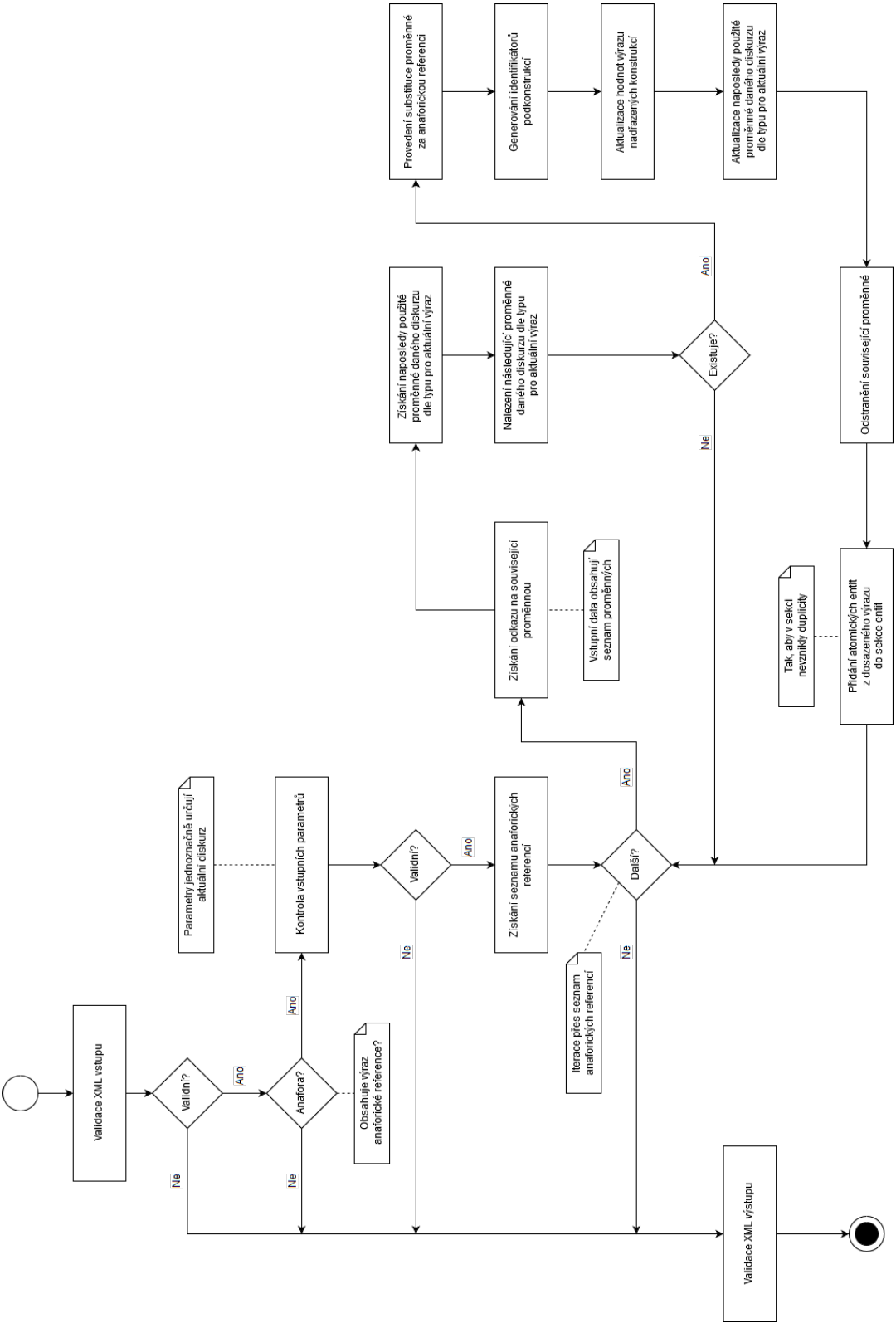
Nyní jsme se dostali až do situace, kdy jsme upravili strukturu stromu konstrukce provedením substituční metody. Je proto potřeba, abychom vhodně upravili některé atributy. Konkrétně musíme upravit identifikátory podkonstrukcí dosazené konstrukce. Jak již bylo zmíněno dříve, při substituci můžeme dosazovat i molekulární konstrukce a dosazením z diskurzní proměnné mohlo dojít k narušení posloupnosti identifikátorů ve stromové struktuře. Podobná situace nastala pro atribut hodnoty, jenž obsahuje proceduru zapsanou v TIL-Scriptu. Substitucí se nám změnil tvar procedury. Je proto třeba zrekonstruovat hodnotu všech konstrukcí nadřazených té, kterou jsme dosadili.

Jakmile máme upravený strom konstrukcí, aktualizujeme informaci, která diskurzní proměnná byla naposledy použita pro celou konstrukci. Následně odstraníme související proměnnou ze sekce `variables`, pokud je to možné (proměnná se již nevyskytuje v jiném konstrukci ve vstupních datech). Do sekce `entities` pak doplníme atomické entity z dosazené diskurzní proměnné.

V předchozích odstavcích je uvedeno, že do diskurzu můžeme ukládat i konstrukce, ve kterých abstrahujeme od konkrétní hodnoty v proceduře s úplným významem. Touto operací získáme



Obrázek 7: Diagram řídícího toku: Aktualizace dynamického diskurzu



Obrázek 8: Diagram řídícího toku: Substituce anafory

Uzávěr (lambda abstrakci), jež očekává argument či argumenty nějakého konkrétního typu. Ku příkladu při dosazení argumentu datového typu `Indiv` do funkce konstruující pravdivostní hodnotu, zda má dané individuum nějakou vlastnost, vznikne ve stromu konstrukce Kompozice lambda abstrakce a argumentu. Pokud je to možné, můžeme proceduru transformovat pomocí operace omezené β-redukce jménem.

Při návrhu algoritmu vznikly další požadavky, zde je pro připomenutí jejich seznam:

1. manuálně vytvořit vstupní data včetně varianty s anaforickým odkazem,
2. diskurz oddělit pomocí `conversation-id` pro možnost vést více konverzací současně,
3. vytvořit schéma XSD pro validaci XML,
4. schopnost rekonstruovat hodnotu a identifikátor konstrukcí po úpravě XML stromu,
5. obohacení diskurzu o konstrukce získané abstrakcí od konkrétní hodnoty ve výrazu (implementovat pro typ `Indiv` s možností rozšíření v budoucnu),
6. provedení omezené β-redukce jménem po dosazení za anaforu (je-li to vhodné).

6 Implementace

Pro implementaci byla zvolena platforma ASP.NET Core od společnosti Microsoft. Jedná se o open source[12] framework[13] pro vývoj moderních webových aplikací a služeb. ASP.NET vytváří webové stránky založené na značkovacím jazyce Hyper Text Markup Language (HTML) v aktuální verzi 5, kaskádových stylech (CSS) a jazyce JavaScript (JS)[9]. Webová technologie byla zvolena na základě úvahy, že agent nasazený na webovém serveru je velmi dobře dostupný v rámci sítě internet. Takový agent může přijímat zprávy nejen prostřednictvím protokolu TCP, ale také v podobě HTTP požadavků.

Platforma .NET Core podporuje celou škálu programovacích jazyků, v této práci jsem pro tvorbu aplikačního kódu použila jazyk C# verze 6. Projekt byl vytvořen v programu Visual Studio 2017. Zdrojové kódy projektu jsou k dispozici v příloze.

Projekt využívá architektonický vzor Model-View-Controller (MVC), jenž rozděluje aplikaci do tří hlavních částí. Použitím tohoto vzoru docílíme logického oddělení komponent systému. Požadavky uživatele vstupují do části Controller zodpovědné za práci s částí Model, ze které můžeme ku příkladu získat výstup pro uživatelem zadaný vstup. Controller také rozhoduje, jaká stránka z části View bude zobrazena uživateli a poskytuje jí potřebná data z části Model [10].

Kromě vzoru MVC bylo v práci využito technologie Razor Pages, což je nová funkcionalita frameworku ASP.NET Core MVP, zjednodušující práci s částí View. Razor Pages se liší oproti čistým MVC projektům mimo jiné tím, že nepoužívá přímo Controller, nýbrž objekty, které dědí ze třídy `PageModel`. Pro jednoduchost je nazývejme *PageModel* daného View. Samotné View obsahuje HTML značky s dodatečnými *tagy* — zkratkami pro možnost využít ve stránce serverovou logiku (např. získání dat ze související třídy PageModel). Samotná třída PageModel je v podstatě ukotvena hned pod souborem View a umožňuje ku příkladu získávat data z databáze pro zobrazení na stránce [11]. Vývoj této části projektu byl inspirován tutoriálem *Introduction to ASP.NET Core* [12].

V ASP.NET Core je rovněž zabudovaná podpora pro Dependency injection (DI). Tímto názvem je označená technika používaná za účelem dosažení nízké úrovně závislostí, tzv. *loose coupling*, mezi objekty a třídami, se kterými spolupracují. V běžné praxi, pokud je objekt závislý na nějaké třídě, vytvoříme její instanci nebo použijeme statickou referenci. DI však umožňuje poskytnout objektu potřebnou třídu jiným způsobem. Nejčastěji třídy deklarují své závislosti v konstruktoru. Tento přístup je známý také jako *constructor injection*. Jeho použitím následujeme *Dependency Inversion Principle*, který lze vyjádřit velmi pěknou poučkou: „Moduly na vyšší úrovni by neměly být závislé na modulech nižší úrovně, obě tyto skupiny by pak měly být závislé na abstrakcích. Abstrakce by neměly záviset na detailech a detaily by měly záviset na abstrakcích“ [13]. Třídy, namísto odkazování se na specifickou implementaci, požadují

[12] Open source označuje software, jehož zdrojové kódy jsou k dispozici volně na internetu.

[13] Framework poskytuje abstraktní generickou funkcionalitu, kterou je možno rozšiřovat o vlastní kód za účelem vytvoření specifické aplikace.

abstrakce (typicky rozhraní), které jim jsou poskytnuty ve chvíli, kdy je třída konstruována. V ASP.NET Core tyto abstrakce poskytuje jednoduchý vbudovaný kontejner[14] reprezentovaný rozhraním `IServiceProvider`.

Aplikaci jsem pojmenovala *TilMan* a skládá se ze dvou projektů. Hlavní projekt `TilMan` je samotná webová aplikace a `TilMan.Tests` projekt definující unit testy. Testování se věnuje Podkapitola 6.8. Hlavní projekt sestává z několika částí. Následující seznam uvádí jednotlivé složky a soubory v projektu:

- `Models` – zapouzdřuje doménovou logiku, reprezentuje stav aplikace a implementuje logiku pro jeho uchování,
- `Pages` – stránky uživatelského rozhraní,
- `Plugins` – obsahuje kód zásuvných modulů a třídy poskytující rozhraní pro práci s nimi,
- `Services` – služby pro komunikaci agenta a zpracování ACL zpráv,
- `Migrations` – definice migrací pro databázi[15],
- `Helpers` – pomocné třídy a metody pro usnadnění práce např. s XML daty, koncovými body síťové komunikace, atd.,
- `Comparers` – kolekce tříd rozšiřujících generickou třídu `EqualityComparer` pro porovnávání objektů (konstrukcí, agentů, aj.),
- soubory v kořenovém adresáři s příponou `.json` či `.config` – konfigurační soubory,
- `Program.cs` – výchozí bod programu, hostuje ASP.NET Core aplikaci, obsahuje metodu `Main`,
- `Startup.cs` – slouží ke konfiguraci služeb a tzv. *pipeline*[16] pro požadavky obdržené aplikací,
- `wwwroot` – složka obsahuje statické soubory (kaskádové styly, JS skripty, apod.).

6.1 Reprezentace XML dat

Předchozí kapitola obsahovala popis vstupních dat a požadavků na ně kladených. Pro potřeby otestování funkcionality implementovaného algoritmu bylo potřeba vytvořit sadu vstupních XML

[14] Kontejnery (anglicky *Containers*) jsou řešení poskytující možnost spouštět SW spolehlivě i za předpokladu, že dochází k jeho přesunutí z jednoho výpočetního prostředí na druhé.

[15] Migrace databáze (také migrace schématu databáze, management změn v databázi) je řízení inkrementálních, vratných změn v relačním databázovém schématu.

[16] Pipeline se využívá pro přesměrování výstupu jedné SW komponenty na vstup druhé komponentě. Lze použít například pro filtrování výsledků.

dat. Především scházela ukázková data obsahující anaforickou proměnnou. Veškerá ručně vytvořená data se nacházejí v projektu `TilMan.Tests` a rovněž v příloze.

Anaforická proměnná je v XML reprezentována volnou proměnnou. Ve výpise 7 je uveden jednoduchý příklad konstrukce výrazu: „He is looking for a parking place.“ Proměnná *he* je v tomto výrazu volná a v uvedeném výpise se jedná o konstrukci `he` typu `variable`, jež *v*-konstruuje datový typ `Indiv`.

```
<construction ID="#2" constructionType="closure" construction="[\w:World [\t:Time [[[’
    Looking_for w] t] he ’Park_Space]]]" type="(((Bool) Time) World)">
  <construction ID="#2#1" constructionType="closure" construction="[\t:Time [[[’
      Looking_for w] t] he ’Park_Space]]" type="((Bool) Time)">
    <construction ID="#2#1#1" constructionType="composition" construction="[[[’
        Looking_for w] t] he ’Park_Space]" type="Bool">
      <construction ID="#2#1#1#1" constructionType="composition" construction="[[’
          Looking_for w] t]" type="(Bool Indiv (((Bool Indiv) Time) World))">
        <construction ID="#2#1#1#1#1" constructionType="composition" construction="[’
            Looking_for w]" type="((Bool Indiv (((Bool Indiv) Time) World)) Time)">
          <construction ID="#2#1#1#1#1#1" constructionType="trivialisation"
              construction="’Looking_for" type="(((Bool Indiv (((Bool Indiv) Time)
              World)) Time) World)"/>
          <construction ID="#2#1#1#1#1#2" constructionType="variable" construction="w"
              type="World"/>
        </construction>
        <construction ID="#2#1#1#1#2" constructionType="variable" construction="t" type
            ="Time"/>
      </construction>
      <construction ID="#2#1#1#2" constructionType="variable" construction="he" type="
          Indiv"/>
      <construction ID="#2#1#1#3" constructionType="trivialisation" construction="’
          Park_Space" type="(((Bool Indiv) Time) World)"/>
    </construction>
  </construction>
</construction>
```

Výpis 7: Příklad Til-Script XML s anaforickou proměnnou

Jakmile takovou konstrukci obdrží aplikace na vstupu, vytvoří si v paměti strom objektů typu `Construction`, jenž ji bude reprezentovat. Následně detekuje její volnou proměnnou a začne hledat vhodnou diskurzní proměnnou pro substituci. Řekněme, že se v diskurzu pro typ `Indiv` nacházela konstrukce `’Tom`. Program nejprve provede definici substituční funkce, čímž dojde k upravení struktury stromu. Pro ukázku uvádím ve Výpise 8 jak vypadá kořenová konstrukce po provedení definice substituce.

```
<construction ID="#2" constructionType="closure"
    construction="[\w:World [\t:Time ^2['Sub ''Tom 'he '[[['Looking_for w] t] he '
        Park_Space]]]]"
    type="((Bool Time) World)">
```

Výpis 8: Příklad Til-Script XML po definici substituce

V dalším kroku se substituční metoda provede a na výstupu získáme uzavřenou konstrukci konstruující výraz s úplným významem. Tím znovu došlo ke změně ve struktuře stromu reprezentující konstrukci. Výsledek provedení substituce znázorňuje Výpis 9.

```
<construction ID="#2" constructionType="closure"
    construction="[\w:World [\t:Time [[['Looking_for w] t] 'Tom 'Park_Space]]]"
    type="(((Bool) Time) World)">
```

Výpis 9: Příklad Til-Script XML po provedení substituce

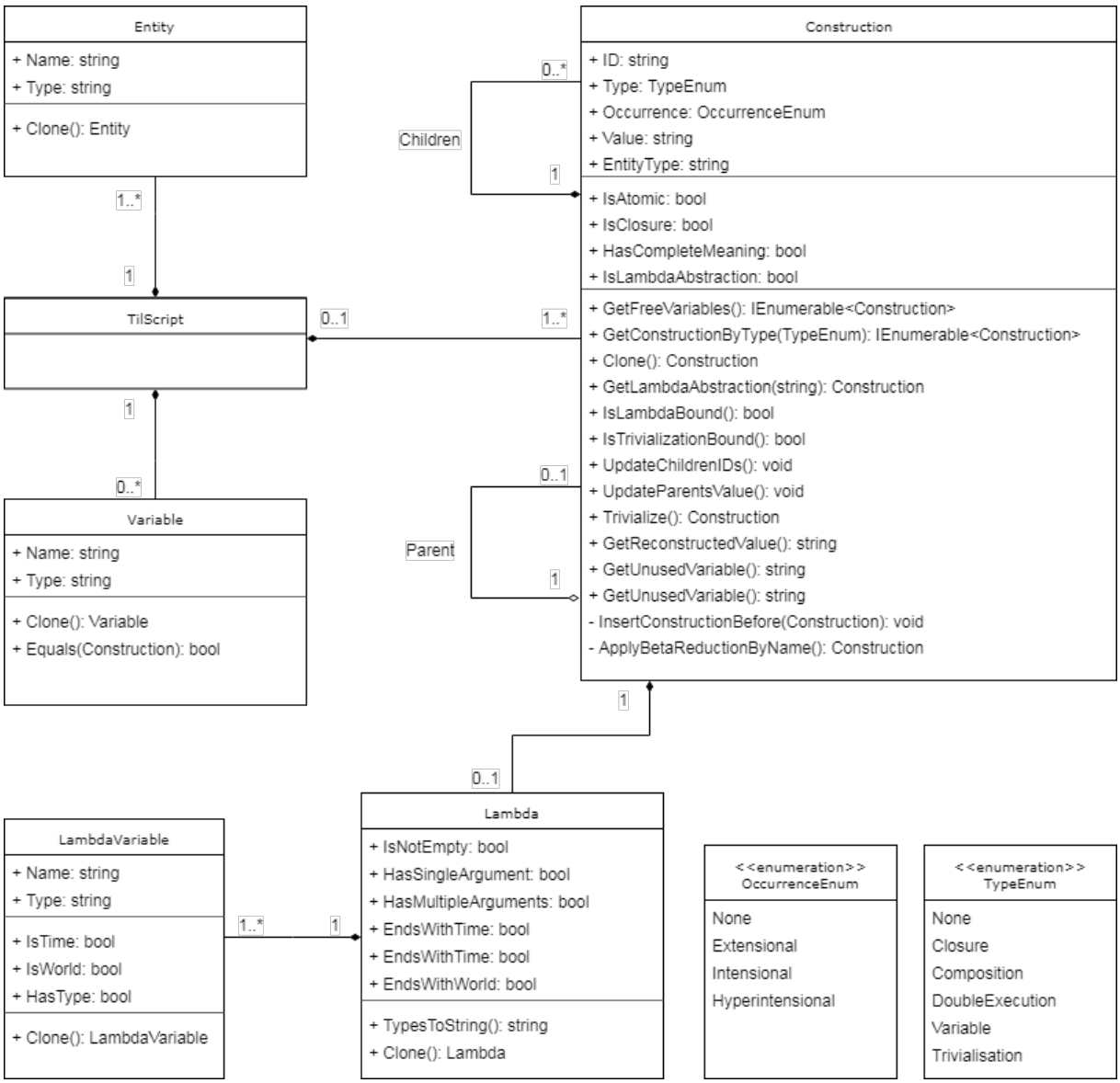
Za účelem eliminace některých lidských chyb, které mohou vzniknout manuálním vytvářením dat, bylo definováno XSD schéma. Verze souboru používaná v aplikaci je k dispozici v přílohách pod názvem `TILscriptXml.xsd`. Toto schéma formálně popisuje jednotlivé elementy v souboru XML. Schéma například kontroluje, že v atributu `constructionType` se nachází pouze jeden z povolených typů dle TIL-Script gramatiky: `trivialisation`, `variable`, `closure`, `n-execution` nebo `composition`.

Na obrázku 9 si můžeme prohlédnout třídní diagram znázorňující jednotlivé části, jež dohromady tvoří celkovou reprezentaci Til-Script XML dat. Po obdržení souboru provede aplikace jeho validaci a pokud má XML požadovanou strukturu, provádí se jeho deserializace, to znamená převod textových dat do podoby objektu. Tuto funkcionalitu zajišťuje pomocná statická třída `XmlHelper`.

Na událost deserializace bylo potřeba navázat další akce. To jsem vyřešila vytvořením třídy s názvem `CustomXmlSerializer` dědící ze třídy `XmlSerializer`, v níž jsem přetížila metodu `Deserialize` a rozšířila ji o callback[17] volání `OnXmlDeserialization`. Po deserializaci TIL-Script XML do podoby objektu `TilScript` tedy dochází k vyvolání tohoto callbacku, v rámci čehož se pro každou konstrukci nastaví reference na jejího rodiče (pokud nějakého má). Dále se provede vytvoření `Lambda` objektů pro každou konstrukci typu `Closure`. Třída `Lambda` slouží pro snažší manipulaci s lambdami a jejich proměnnými.

Na výstupu deserializace jsme tedy obdrželi instanci třídy `TilScript`, kterou nyní můžeme předat modulu pro zpracování anafory.

[17] Callback je libovolný spustitelný kód, jenž předáváme jako argument jiné části kódu. Očekáváme, že tato část „zavolá zpět“ (spustí) argument v danou chvíli.



Obrázek 9: Třídní diagram: Objektová reprezentace TIL-Script XML dat

6.2 Zásuvný modul pro řešení anafory

Tato podkapitola je věnovaná popisu souboru tříd pro práci se zásuvnými moduly, tzv. *plugin management*. Také je zde popsána funkce modulu pro práci s anaforou.

V Kapitole 5 si můžeme přečíst, že systém pro řešení anafory má být koncipován formou zásuvného modulu. V návrhu algoritmu již bylo popsáno rozhraní, jež budou moduly využívat. Pro práci s moduly však stále potřebujeme definovat způsob, jak moduly v systému registrovat a jak s nimi pracovat. Systém pro řešení zásuvných modulů byl inspirován open source projektem Internet of Things (IoT) s názvem *Temperature Station* [14].

Veškerý kód související se zásuvnými moduly se v projektu nachází ve složce `Plugin`. Pro moduly je specifikováno rozhraní `IPlugin`. Samostatný modul je pak třída implementující toto rozhraní. Druhé rozhraní je nazváno `IPluginProvider` a poskytuje přístup k modulům a jejich atributům. O registraci modulů se stará implementace ve třídě `PluginsLoader`. Pokud chceme přidat další modul, stačí nahrát nebo vytvořit novou třídu. `PluginsLoader` se postará o její načtení z *Assembly*[18] a registraci v aplikaci.

`PluginAttribute` je třída reprezentující atributy modulů a dědící ze třídy `Attribute`. Konkrétně se jedná o atributy `Order` a `Name`. Nutno podotknout, že atribut pořadí (`Order`) určuje posloupnost, v jaké se budou moduly vykonávat. Proto je třeba dbát na to, aby hodnota pořadí byla pokud možno unikátní. Pokud by dva moduly měly stejnou hodnotu pořadí, nebude jasné, který z nich se má vykonat dříve. Jelikož moduly mohou využívat constructor injection pro přístup ke službám či databázi, je třeba jednotlivé moduly zaregistrovat v kontejneru `IServiceCollection` při spuštění aplikace. O toto se stará poslední součást managementu zásuvných modulů pojmenovaná `PluginExtensions`.

Jednotlivé moduly jsou třídy se sufixem `Plugin`. Následující text se věnuje modulu pro řešení anafory pojmenovaném `AnaphoraSubstitutionPlugin`. Při inicializaci modulu třída obdrží reference na abstrakce pro přístup ke konfiguraci aplikace, databázi a také pro logování. Modul využívá tři definice z konfiguračního souboru:

1. `DiscourseCandidatesEnabled` určuje, zda se po doplnění anafory ukládají výsledné konstrukce jako tzv. *kandidáti*,
2. `DiscourseVariablesPerEntityType` definuje maximální počet konstrukcí, které jsou uloženy pro jeden datový typ v daném diskurzu,
3. `EntityTypesForLambdaAbstractionList` reprezentuje seznam datových typů, od kterých budeme abstrahovat při tvorbě diskurzu.

Slovem *kandidát* označujeme konstrukci ukládanou do databáze, v níž byla provedena substituce za anaforickou proměnnou. V teoretické části popisující algoritmus anafory bylo vysvětleno,

[18] Assembly se nazývají hlavní stavební bloky .NET Core aplikací. Jedná se o kolekci typů a zdrojů, jež jsou sestaveny, aby společně pracovaly a vytvořily logickou jednotku funkcionality.

proč nemusí být první dosazená diskurzní proměnná ta správná. Pokud nejsme s výsledkem substituce spokojeni a chceme získat novou konstrukci dosazením další diskurzní proměnné, můžeme aplikaci znovu předat stejnou konstrukci. V takovém případě jsou konstrukce uložené s příznakem kandidáta smazány, je provedena substituce a výsledná konstrukce a konstituenty jsou uloženy opět jako kandidáti.

Pokud při aktualizaci diskurzu dojde k překročení maximálního počtu diskurzních proměnných pro daný typ, dojde k odstranění těch nejstarších tak, aby jejich počet nepřekračoval tento limit.

Modul očekává, že před jeho použitím mu bude nastaven parametr `ConversationId`. Tento parametr jednoznačně identifikuje konverzaci a tou je ohraničen i daný diskurz. Samotný algoritmus již byl znázorněn na Obrázcích 7 a 8. Metoda pro výběr vhodné diskurzní proměnné v nich byla poněkud zjednodušena a proto se na ni teď podíváme důkladněji.

Otevřená konstrukce může obsahovat jednu nebo více anaforických proměnných. Pokud konstrukce obsahuje pouze jedinou, pak je získání další diskurzní proměnné jednoduché. Při první substituci dosadíme nejnovější proměnnou daného typu, při opakování substituce dosadíme druhou nejnovější, atd. Pokud se však ve výrazu nachází více anaforických proměnných, je potřeba mít jistotu, že systém je schopen vrátit všechny možné kombinace diskurzních proměnných.

Pro ukázku nyní předpokládejme, že v diskurzu mohou být maximálně 3 položky pro každý datový typ. Uvažujme situaci, kdy máme v konstrukci tři anaforické proměnné a, b a c stejného typu. Každé proměnné přiřadíme pořadí a učiníme tak zprava doleva, přičemž začínáme počítat od nuly. Proměnným tedy odpovídá pořadí 2, 1 a 0. Nyní každé proměnné přiřadíme index reprezentující diskurzní proměnnou, která za ni bude substituována. Pokud máme v diskurzu maximálně tři položky, lze si přiřazení představit jako sekvenci čísel v rozmezí $0 - 2$. Poté provádíme postupnou inkrementaci od proměnné nejvíce vpravo. V Tabulce 6 je znázorněno prvních 16 kombinací.

volná proměnná	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
a	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
b	0	0	0	1	1	1	2	2	2	0	0	0	1	1	1	2
c	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0

Tabulka 6: Přiřazování diskurzních proměnných v konstrukci se třemi volnými proměnnými (prvních 16 iterací).

Algoritmus pro přiřazení je zachycen pseudokódem ve Výpise 10. Na vstupu se očekává seznam volných proměnných (`ref`). V pseudokódu byla použita tečková notace pro přístup k vlastnostem, tzv. *properties*, dané reference. Naposledy použitá diskurzní proměnná je v kódu nazvána `Previous`. Diskurzní proměnnou, která bude použita nyní, označuje property `Next`. Pomocí property `Count` získáme počet volných proměnných. Také jsou zde použity dvě blíže nespecifikované metody. První `GetNext()` vrátí následující diskurzní proměnnou v závislosti na tom, jakou jsme

naposledy použili (pokud vůbec nějakou). Druhá metoda `GetFirst()` pak vrátí první (nejnovější) diskurzní proměnnou v seznamu.

```
GetNextVariables(ref):
  nextFound := false
  for i := ref.Count to 0 do
    if nextFound then
      ref[i].Next := ref[i].Previous
      continue
    end
    if ref[i].Previous exists then
      ref[i].Next := ref[i].GetNext()
    end
    if ref[i].Next exists then
      nextFound := true
    else
      if ref[i + 1] exists then
        ref[i].Next := ref[i].GetFirst()
        if ref[i].Next exists then
          continue
        end
      return false
    end
  end
  return true
```

Výpis 10: Pseudokód algoritmu pro hledání vhodných diskurzních proměnných

Reprezentace konstrukcí byla popsána v předchozí kapitole. Z pohledu algoritmu pro řešení anafory bylo potřeba ve třídě `Construction` vytvořit metody pro práci s konstrukcemi, např. výběr konstrukcí dle nějakého kritéria, vytvoření nových konstrukcí, kontrola konstrukce na různé podmínky, modifikace stromu konstrukcí, získání rekonstruované hodnoty vlastností konstrukce, atd. V kódu je implementováno velké množství různých metod a zde si uvedeme několik z nich. Jedná se o ty, jež jsou zajímavé v souvislosti s řešením anafory. V případě hlubšího zájmu je v příloze k dispozici programátorská dokumentace, v níž jsou popsány všechny.

Metoda `GetFreeVariables` z konstrukce vybere všechny volné proměnné a na výstupu vrátí jejich enumerátor[19]. Ve výpise 11 si můžeme prohlédnout její zdrojový kód.

```
public IEnumerable<Construction> GetFreeVariables()
{
    return GetAtomicConstructions().Where(c => c.IsVariable && !c.IsTrivializationBound
        () && !c.IsLambdaBound());
}
```

Výpis 11: Metoda pro získání enumerátoru volných proměnných v konstrukci

[19]Enumerátor podporuje jednoduchou iteraci přes kolekci objektů specifikovaného typu.

V kódu je použita metoda `GetAtomicConstructions`, jež vrací enumerátor všech atomických konstrukcí. Funkce `Where`[20] pak filtruje danou sekvenci v závislosti na predikátu definovaném v argumentu. Ve výpise použitý predikát říká, že atomická konstrukce musí být typu `Variable`, ale nesmí být vázaná Trivializací ani lambda vázaná. Kontrola, zda je proměnná vázaná Trivializací, se provádí rekurzivním průchodem stromem konstrukcí od proměnné směrem nahoru, přičemž každý rodič v této větvi je zkontrolován na podmínku, zda je typu `Trivialization`.

Druhá kontrola sestává také z procházení rodičů proměnné prostřednictvím rekurze. Každý rodič (`Parent`) je však testován na sérii podmínek. Výpis zdrojového kódu 12 znázorňuje jednotlivé podmínky. Na vstupu jsou dva argumenty, řetězec `value` obsahuje hodnotu konstrukce a řetězec `entityType` typ touto konstrukcí konstruovaný. Pokud se dříve než k lambda funkci dostaneme k rodiči typu `Trivialization`, pak proměnná není lambda závaná (vrací se `false`). V tomto případě je totiž vázaná Trivializací. V opačném případě se kontroluje, zda daná konstrukce obsahuje zmíněnou proměnnou. Tato podmínka je zde proto, aby šlo metodu použít i pro případy, kdy nepostupujeme rekurzí přímo od konkrétní proměnné, a pro tento scénář není až tak zajímavá. Následně se provádí kontrola, zda je daný rodič typu `Closure` a zda se mezi lambda proměnnými nachází nějaká s názvem a typem, jež odpovídají zadaným argumentům. Pokud ano, vrátí se nám hodnota `true`. Není-li splněna žádná z uvedených podmínek, pak se pokračuje s dalším nadřazeným uzlem, existuje-li takový. Může se stát, že proměnná není vázaná lambdou a rovněž není pod trivializací. Pak dostaneme potvrzeno, že se jedná o volnou proměnnou.

```
private bool IsLambdaBound(string value, string entityType)
{
    if (IsTrivialization)
        return false;
    if (!ContainsVariable(value, entityType))
        return false;
    if (IsClosure && Lambda.Variables.Any(v => v.Name.Equals(value) && v.Type.Equals(
        entityType)))
        return true;
    if (Parent != null)
        return Parent.IsLambdaBound(value, entityType);
    return false;
}
```

Výpis 12: Metoda pro kontrolu, zda je proměnná lambda vázaná

Poslední metoda, kterou v této podkapitole zmíním, je `InsertConstructionBefore`. Uvádím ji zde proto, že řeší jeden z bodů návrhu a to konkrétně schopnost rekonstruovat hodnotu a identifikátor konstrukcí po úpravě XML stromu. Metoda se používá v různých situacích, ku příkladu po provedení lambda abstrakce, při definici substituce, atd. Je privátní, to znamená, že ji

[20] Jedná se o funkci z knihovny Language Integrated Query (LINQ), což je .NET Framework komponenta od společnosti Microsoft, která přidává kapabilitu nativního dotazování nad daty v jazycích .NET.

nelze volat z jiné třídy. Toto omezení je zde z důvodu zapouzdření. To znamená proto, že sama třída `Construction` by se měla starat o aktualizaci stromu konstrukcí a nikdy by tento podnět neměl přijít z jiné třídy.

Kód ve Výpise 13 provede vložení konstrukce v argumentu funkce nad pozici aktuální konstrukce ve stromové struktuře. Vkládaná konstrukce obdrží `ID` a také odkaz na rodiče té aktuální. Poté je aktuální konstrukce přidána jako potomek konstrukce vkládané. Následně se provede rekonstrukce hodnoty vkládané konstrukce. Existuje-li rodič aktuální instance, pak u něj vyměníme referenci na aktuální instanci za referenci na vkládanou konstrukci. Rodičem aktuální instance se stává vkládaná konstrukce. Na závěr metody se provede aktualizace hodnot konstrukcí pro všechny rodiče ve větvi počínaje vloženou konstrukcí a také aktualizace identifikátorů od potomků vložené konstrukce až k listům.

```
private void InsertConstructionBefore(Construction construction)
{
    construction.ID = ID;
    construction.Parent = Parent;
    construction.Constructions.Add(this);
    construction.Value = construction.GetReconstructedValue();
    if (Parent != null)
    {
        Parent.ReplaceChild(this, construction);
    }
    Parent = construction;
    construction.UpdateParentsValue();
    construction.UpdateChildrenIDs();
}
```

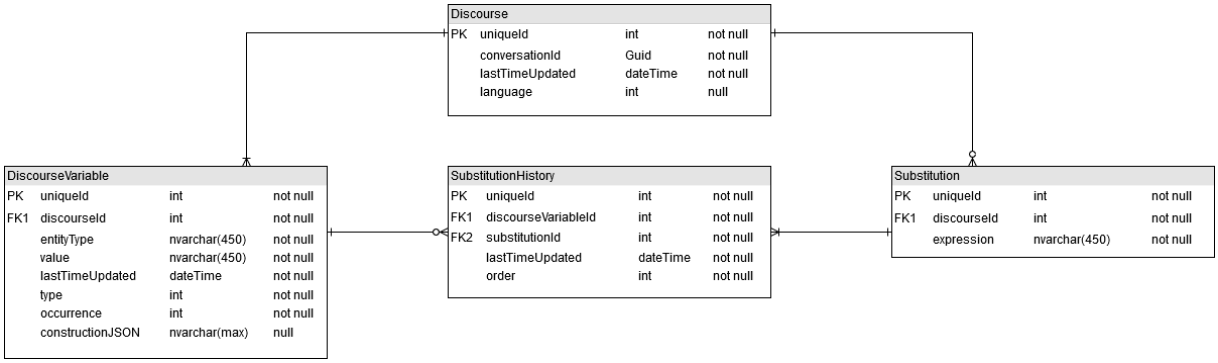
Výpis 13: Metoda pro úpravu struktury stromu konstrukcí při vložení konstrukce v argumentu nad aktuální instanci

Na závěr jen poznamenejme, že v TIL-Script XML mohou být analyzovány kontexty. V praxi to znamená, že je k dispozici hodnota atributu `occurrence`. Modul pro řešení anafory s touto hodnotou nijak nepracuje a proto ani není povinná v XSD definici. Na základě domluvy během konzultací je však zajištěno, že kontext zůstane po substituci konstrukce stejný, jaký byl v době uložení konstrukce do diskurzu. Kontexty nejsou dále nijak kontrolovány.

6.3 Databáze a logování

Pro uchování dat byla zvolena relační databáze. V průběhu lokálního vývoje jsem používala *LocalDB* obsahující minimální množství souborů potřebných pro běh *SQL Server Database Engine*[21]. Jedná se o instanci systému SQL Server Express[22], jež je k dispozici prostřednictvím

[21] SQL Server Database Engine je klíčová služba pro ukládání, zpracování a zabezpečení dat.
[22] SQL Server Express je relační databázový management systém, jež lze volně stáhnout, distribuovat a používat.



Obrázek 10: ER diagram databáze diskurzů

Integrated development environment (IDE) Visual Studio 2017. Pro práci s databází se využívá *Entity Framework* v aktuální verzi 6. Databázová struktura byla definována přístupem *Code First*, kdy došlo nejprve k vytvoření tříd reprezentujících databázový model. Další definice lze volitelně přidat pomocí atributů ke třídám nebo třídním proměnným, např. identifikace primárního či cizího klíče, formát zobrazení uložené informace, nebo pokyn, aby některá property nebyla do databáze mapovaná.

V databázi se nacházejí dvě hlavní skupiny tabulek. První souvisí s diskurzem, ukládáním diskurzních proměnných a jejich substitucí v konstrukcích. Na Obrázku 10 je zobrazen na *Entity-Relationship* diagramu (ERD). Tabulka `Discourse` jednoznačně identifikuje daný diskurz za použití sloupců `conversationId` a jazyka `language`, jenž však nemusí být definovaný.

Každý diskurz sestává z jedné či více diskurzních proměnných (`DiscourseVariable`), jež jsou unikátní kombinací hodnot ve sloupcích `entityType`, `value` a cizím klíčem `discourseId`. Sloupec `constructionJSON` zachycuje stromovou strukturu a atributy konstrukcí ve formátu JSON, jež jsou potomky konstrukce uložené v diskurzní proměnné. Pokud jsou data načtena z databáze, převede se tento formát na objektovou reprezentaci konstrukcí tak, jak se v systému normálně používá. Položky z číselníků (`Enum`) jsou v databázi uloženy jako hodnoty typu `int` (integer). Při načtení dat z databáze se provádí jejich převod na danou hodnotu číselníku a objekty reprezentující jednotlivé záznamy již pracují pouze s číselníky. Jako příklad mohu uvést číselník `TypeEnum`, jehož položky odpovídají jednotlivým hodnotám, jakých může nabývat atribut `constructionType` v TIL-Script XML.

V rámci diskurzu jsou definované také položky substituce (`Substitution`), jejichž funkce byla objasněna v předchozí podkapitole. Mezi tabulkami `Substitution` a `DiscourseVariable` se nachází vazební tabulka `SubstitutionHistory`. V této tabulce se nachází informace o tom, jaká diskurzní proměnná byla naposledy použita pro kterou substituci včetně uvedení pořadí proměnné v dané konstrukci.

Pozorného čtenáře mohlo napadnout, že z pohledu uchovávání dat, je systém pro řešení anafory velmi citlivý. Pokud by měl někdo možnost upravovat údaje v databázi, velmi snadno by nepromyšlenými zásahy do hodnot položek v databázi mohl narušit správnou funkci mo-

Obrázek 11: ER diagram databáze agentů

dulu. Ku příkladu by pak mohlo dojít k dosazování diskurzních proměnných pro jinou konverzaci, apod. Z tohoto důvodu se v aplikaci k databázi přistupuje prostřednictvím jediné třídy `ApplicationContext`. S daty se v aplikaci pracuje pomocí properties generického datového typu `DbSet<T>`, a některé tyto properties jsou skryté modifikátorem `private`. Lze je pak upravovat pouze speciálními medotami. Tento přístup je použit například u tabulek `Discourse` nebo `Substitution`.

Druhá skupina tabulek řeší reprezentaci agentů. ER diagram se skládá pouze ze dvou tabulek a lze si jej prohlédnout na Obrázku 11. Jeden agent může mít přiřazeno více adres, prostřednictvím kterých ho můžeme kontaktovat. Pokud se některá stane nedostupnou, můžeme pak použít jinou. Adresa je dána typem transportu (`transportType`), jenž může v současnosti nabývat hodnot: `Socket`, `Http` či `NotDefined`. Hodnota ve sloupci `locator` je pak konkrétní adresa daného typu transportu zapsaná v podobě řetězce. Pro typ `Socket` by měla jeho hodnota odpovídat formátu `<IP Adresa>:<Port>`. Naopak pro typ `Http` by se mělo jednat o řetězec, z něhož lze vytvořit absolutní Uniform Resource Identifier (URI).

Ve specifikaci požadavků bylo zmíněno, že agent má disponovat schopností vrátit seznam jmen agentů, které zná. Třída `ApplicationContext` z tohoto důvodu disponuje metodou pro získání seznamu všech agentů z databáze. Postupně při komunikaci s agenty si tedy aplikace ukládá adresy agentů. Tyto adresy nejsou získávány ze samotné komunikace, ale z obsahu ACL zprávy, jež agent obdrží. Blíže se tomuto tématu věnuje Podkapitola 6.6.

Neustále je zmiňováno, že data budou v databázi přibývat. Aby však nedocházelo k neustálému navyšování kapacity databáze, je možné nastavit plán čištění databáze (`CleanupTask`). Tuto funkcionalitu poskytuje open source balíček *Hangfire*[16]. Balíček vykonává akce na pozadí v procesu aplikace podobně jako *cron*[23]. Při práci s aplikací si uživatel ani nevšimne, že probíhá nějaký nastavený úkol. Jak často čistění probíhá a kolik dní zpětně si aplikace má pamatovat položky v diskurzu, lze nastavit v konfiguraci programu. V rámci úkolu `CleanupTask` dochází k následujícímu:

1. Pro jednotlivé agenty se aplikace pokusí připojit k jejich primární adrese. Pouze v případě, že se to nepodaří, je adresa odstraněna z databáze,

2. nemá-li agent žádné adresy (tzn. na žádné adrese se jej nepodařilo kontaktovat), je agent odebrán z databáze agentů,

[23]SW utilita cron plánuje úkoly, jež provádí v daném čase. Používá se v operačních systémech založených na Unixu.

3. smazání záznamů z tabulky `Discourse` starší než daný počet dnů, včetně souvisejících záznamů z tabulek `DiscourseVariable`, `Substitution` a `SubstitutionHistory`.

Důležité informace o provedeném čištění jsou logovány. ASP.NET Core framework poskytuje API pro logování, jež podporuje řadu způsobů, jak logy poskytnout uživateli. Mezi zabudované způsoby patří logování do konzole, do služby *EventLog* v prostředí Windows, apod. Rovněž umožňuje napojit logovací frameworky třetích stran. V této práci je použit balíček *NLog*, což je open source platforma pro .NET aplikace s bohatými možnostmi v oblasti směrování a managementu logů[17]. Pro účely této práce se využívá logování do souboru. Balíček NLog se postará o formátování logů a rozdělení do souborů pro každý den zvlášť.

Za použití tzv. *Scaffolding* techniky bylo specifikováno, jak může být využita aplikační databáze. Kompiler a framework pak použili tuto specifikaci spolu s předdefinovanými šablonami („scaffold“) za účelem vygenerování kódu, jenž byl využit pro zobrazení záznamů z databáze prostřednictvím přehledného uživatelského rozhraní.

6.4 Síťová komunikace

Jedním požadavkem ze specifikace je schopnost aplikace přijmout zprávy na protokolu TCP. Původní implementace agenta zmíněná v Kapitole 5 používala pro komunikaci User Datagram Protocol (UDP), proto i aplikace TilMan ze začátku obsahovala tento způsob komunikace. Později se na konzultacích domluvilo, že komunikace bude probíhat prostřednictvím TCP protokolu. Jelikož již UDP část byla hotova, rozhodla jsem se ji v aplikaci ponechat.

Třídy pro síťovou komunikaci jsou vždy dvě. První pro příchozí komunikaci (`Receiver`) implementují rozhraní `IReceiver` a druhá pro odchozí komunikaci (`Sender`) implementující rozhraní `ISender`. Pro příchozí komunikaci zavoláme pro instanci třídy `TcpReceiver` metodu `BeginReceive` s jedním argumentem obsahující číslo portu, na němž chceme poslouchat. Metoda inicializuje klienta, který implementuje Berkeleyho socket[24], aby začal poslouchat na daném portu a asynchronně přijímal zprávy. Na základě konzultací bylo dohodnuto, že se pro TCP komunikaci budou používat porty v rozsahu 49152-65535. Tyto byly označeny společností Internet Assigned Numbers Authority (IANA) jako lokální či privátní a nemohou být přiřazeny jinému účelu[19].

Dále bylo domluveno, že zprávy budou zakódovány ve formátu UTF-8[25], čímž bude podpořena i kompatibilita s aktuální verzí TIL-Script specifikace. Maximální velikost TCP packetu je stanovena na 1500 bajtů. Pokud se snažíme zakódovat větší zprávu, pak dojde k jevu, který se nazývá *fragmentace*. Zpráva je rozdělena do více packetů, které se postupně posílají. Pokud chceme takovou zprávu v aplikaci přijmout, potřebujeme předem znát:

a) Přesný počet bajtů, které dohromady tvoří zprávu, nebo

[24] Berkeleyho socket je knihovna poskytující API pro internetové sockety a sockety Unixové domény, které se používají pro komunikaci mezi procesy.

[25] Unicode Transformation Format (UTF) je způsob kódování řetězců znaků do pole bajtů.

b) dohodnout se na speciálním znaku, jenž bude zprávu vždy ukončovat.

Domluveno bylo použití speciální značky `EOF` z ASCII tabulky. Její hodnota v hexadecimálním zápisu je `0x03`.

Třída `TcpSender` je v případě potřeby instanciována a prostřednictvím metody `Send` odesílá zprávu v kódování UTF-8 z náhodného portu na adresu příjemce. Zpráva je v případě potřeby také rozdělena, o to se však stará až vnitřní implementace metody `BeginSend` ze třídy `Socket`.

Inspirací při implementaci mi byl online tutoriál *Asynchronous Server Socket Example*[18].

6.5 Parsování ACL zpráv

Po obdržení paketů nebo HTTP requestu je potřeba porozumět jeho obsahu. V Podkapitole 4.2 byla popsána struktura ACL zprávy. Pro snažší práci s ACL zprávou je vhodné ji převést do formy objektu. Za tímto účelem byly implementovány třídy pro provedení lexikální analýzy a následně syntaktické analýzy rekurzivním sestupem. Gramatika ACL zprávy je popsána v dokumentu FIPA reprezentace ACL zprávy v podobě řetězce (*SC00070I*). Je napsaná v EBNF, která pracuje i s regulárními operátory. S ohledem na specifikaci požadavků byly některé části rozšířeny. Například adresy agenta mohou být nejen URL, ale také tvaru `<IP Adresa>:<Port>`.

Lexikální analýza rozdělí řetězec znaků na tzv. *lexémy*, což jsou lexikální jednotky (např. klíčová slova, čísla, atd.) a odstraní bílé znaky ze zdrojového řetězce. Lexémy jsou reprezentovány ve formě *tokenů*. Implementace byla inspirována článkem na blogu Jacka Vanlightlyho[20].

Hlavní součástí lexikální analýzy je statická třída `Tokenizer`, v níž je inicializován seznam obsahující definované tokeny. Ty jsou jednoznačně identifikovatelné pomocí hodnoty číselníku `TokenTypeEnum`, regulárního výrazu a čísla určujícího prioritu tokenu. Ve výpise 14 jsou uvedeny některé z definic tokenů. Priorita funguje tím způsobem, že pokud během procházení řetězce začínají na stejné pozici různé tokeny, vybere se nakonec ten s nejvyšší prioritou. Na uvedeném příkladu má nejvyšší prioritu token typu `String`. Je to tak z toho důvodu, že v ACL zprávě se `Content` neboli obsah zprávy nachází mezi znaky uvozovek. Pokud by se někde mezi uvozovkami nacházela další uvozovka, je potřeba ji tzv. *escapovat*[26]. Při parsování pak chceme vybrat co nejdelší řetězec začínající a končící uvozovkami. Regulární výraz pro token typu `String` definuje, že mezi dvěma uvozovkami se mohou vyskytovat pouze znaky, které nejsou uvozovky v počtu nula a více znaků. Zároveň znak uvozovek je povolen pouze, pokud je escapovaný.

Ve třídě se nachází také metoda `Tokenize`, která je vstupem pro celý algoritmus lexikální analýzy. Tato metoda vrací enumerátor objektů datového typu `AclToken`. Tento objekt má dvě properties: typ tokenu `TokenType` datového typu `TokenTypeEnum` a řetězec `Value` (hodnota tokenu). Pomocí číselníku `TokenTypeEnum` lze s tokeny pohodlně pracovat při následném rekurznivním sestupu.

[26]Escapování se slangově nazývá uvozování znaků se speciálním významem v textových řetězcích vybraným znakem (či znaky), jenž je označován jako tzv. escape character. Nejčastěji se jedná o zpětné lomítko „\“. Takto prefixovaný znak již nemá v řetězci speciální význam.

```
private static List<TokenDefinition> _tokenDefinitions = new List<TokenDefinition>
{
    new TokenDefinition(TokenTypeEnum.Inform, "inform", 3),
    new TokenDefinition(TokenTypeEnum.Word, $"[^\x00-\x20\\(\\)#0-9\\-@][^\x00-\x20
        \\(\\)]*", 4),
    new TokenDefinition(TokenTypeEnum.String, $"\"([^\"]|\\\")*\"", 1),
    new TokenDefinition(TokenTypeEnum.Integer, $"(\\+|\\-)?\\d+", 5),
    new TokenDefinition(TokenTypeEnum.IPv4, $"
        ((25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])\\.)
        {3}(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])", 3),
};
```

Výpis 14: Lexikální analýza – příklad některých definicí tokenů

Syntaktická analýza rekurzivním sestupem je definovaná ve statické třídě `RecursiveDescent`. Rekurzivní sestup zahájíme zavoláním metody `Execute`, které dáme na vstupu enumerátor s objekty `AclToken`. Následně postupujeme token po tokenu, ověřujeme, zda má zpráva očekávanou syntaxi a zároveň doplňujeme hodnoty z tokenů do patřičných properties nové instance třídy `AclMessage` reprezentující ACL zprávu jako objekt v paměti. Výpis kódu 15 obsahuje výňatek z FIPA gramatiky pro ACL zprávy a to konkrétně definici agenta `AgentIdentifier` a `Word` je lexikální pravidlo.

```
AgentIdentifier = "(" "agent-identifier"
                    ":name" Word
                  [ ":addresses" URLSequence ]
                  [ ":resolvers" AgentIdentifierSequence ]
                  ( UserDefinedParameter Expression )* ")".

Word            = [~ "\0x00" - "\0x20", "(", ")", "#", "0" - "9", "-", "@"]
                  [~ "\0x00" - "\0x20", "(", ")"]*.
```

Výpis 15: Část gramatiky ACL zprávy dle specifikace FIPA – definice agenta

Zaměřme se například na parsování jména agenta. Všimněme si, jak FIPA specifikaci definuje jméno agenta. U lexikálních pravidel hranaté závorky uzavírají množinu znaků a pomlčka značí rozsah (např. `["A"- "Z"]` zahrnuje všechny znaky ASCII abecedy od písmene velké „A“ až po písmeno velké „Z“). Pokud je vlnovka první znak, pak označuje doplněk množiny znaků. Navíc oproti FIPA specifikaci bylo na konzultacích dohodnuto, že jméno agenta bude mít tvar `<IP Adresa>:<Port>`. Následující Výpis 16 znázorňuje související část syntaktické analýzy, jež počítá s možností, že jméno agenta bude dle FIPA specifikace, nebo bude ve formě IP adresy verze 4 a portu, či IP adresy verze 6 a portu.

Pokud při parsování dojde k výskytu neočekávaného tokenu, je vyhozena výjimka typu `UnexpectedTokenTypeException`.

```
private static string GetAgentName(IEnumerator<AclToken> enumerator)
{
    if (enumerator.Current.TokenType == TokenType.Word)
    {
        return enumerator.Current.Value;
    }
    else if (enumerator.Current.TokenType == TokenType.IPv4 || enumerator.Current.
        TokenType == TokenType.IPv6)
    {
        return GetIPAndPort(enumerator);
    }
    else
        throw new UnexpectedTokenTypeException("agent name", enumerator.Current.
            TokenType);
}

private static string GetIPAndPort(IEnumerator<AclToken> enumerator)
{
    if (enumerator.Current.TokenType != TokenType.IPv4 && enumerator.Current.TokenType
        != TokenType.IPv6)
        throw new UnexpectedTokenTypeException($"{TokenType.IPv4} or {TokenType.IPv6}",
             enumerator.Current.TokenType);

    var address = enumerator.Current.Value;

    enumerator.MoveNext();
    if (enumerator.Current.TokenType != TokenType.Port)
        throw new UnexpectedTokenTypeException(TokenType.Port, enumerator.Current.
            TokenType);

    int.TryParse(enumerator.Current.Value.Substring(1), out int port);
    return $"{address}:{port}";
}
```

Výpis 16: Syntaktická analýza rekurzivním sestupem – příklad parsování jména agenta

6.6 Rozhodovací modul

Jakmile máme k dispozici objektovou reprezentaci ACL zprávy, předáme ji na vstupu metody `Receive` instanci třídy implementující rozhraní `IAclMessageHandler`. Ta vrací enumerátor `AclMessage` objektů, jež reprezentují odpovědi na danou zprávu. Tato metoda je navíc implementována jako tzv. *generátor*. Jedná se o speciální postup, jakým můžeme kontrolovat chování iterovaného cyklu. Při běžném postupu ve funkci nejprve inicializujeme a naplníme nějaké pole všemi hodnotami. Na výstupu funkce pak vracíme takto naplněné pole. Oproti tomu generátor dodává vždy jednu hodnotu v danou chvíli, čímž vyžaduje méně paměti a umožňuje volající metodě začít s hodnotou okamžitě pracovat bez potřeby čekat na zbylé hodnoty. Implementace generátoru je velmi podobná normálnímu iterátoru, ale namísto klíčového slova `return` se použije `yield return`, případně `yield break` pro ukončení iterace. Volaná metoda, v tomto případě metoda `Receive`, se pak použije v iteraci. Konkrétně se jedná o iteraci, během níž dochází k postupnému odesílání ACL zpráv reprezentujících odpověď či odpovědi.

V samotné metodě `Receive` se nejprve předpřipraví šablona odchozí zprávy. Nastavíme odesílatele (property `Sender`) a hodnotu specifikující na jaký typ performativu odpovídáme (property `InReplyTo`). Některé příchozí zprávy mohou mít přímo specifikováno, jakým agentům mají být zaslány odpovědi (property `ReplyTo`). Pokud je to uvedeno, pak se tyto agenty nastaví jako příjemci odpovědí (seznam příjemců reprezentuje property `Receiver`). Není-li to však specifikováno, stane se příjemcem pouze odesílatel původní zprávy. Na závěr nesmíme zapomenout na nastavení property `ConversationId`. Pokud pokračujeme v dané konverzaci, nastavíme stejné `ConversationId`, jaké bylo použito v příchozí zprávě.

V této metodě rovněž dochází k rozhodnutí, jak agent na příchozí zprávu odpoví. Provádění rozhodnutí je implementováno také jako generátor. Na jeho výstupu jsou pak datové struktury pojmenované `Reply` obsahující hodnoty parametrů `Performative`, `Language`, `ReplyBy` a především `Content`. Postupně, vždy jakmile máme k dispozici jeden z výstupů, jej vložíme do předpřipravené odchozí zprávy, kterou následně rozesíláme jednotlivým příjemcům. Při odesílání ukládáme do databáze agenty (příjemce).

Samotné rozhodování je v aktuální verzi aplikace implementováno ve třech souborech `Decide`, `DecideInform` a `DecideQueryRef`. Všechny tři obsahují část třídy `Decide`, proto je v kódu označená jako `partial`. To znamená, že všechny tři soubory jsou v programu chápány jako jeden celek. Nejprve je kód rozvětven dle typu performativu příchozí zprávy. Aplikace umí v současné době zpracovat dva typy performativů, jimž odpovídají sufixy zmíněných souborů. Obdrží-li agent jiný typ performativu, než jaký umí zpracovat, pak jej ignoruje.

Je-li typ performativu příchozí zprávy `inform`, agent provede deserializaci TIL-Script XML do podoby objektu `TilScript`, jenž následně zpracují zásuvné moduly. V rámci modulu řešení anafory tedy aktualizuje diskurz a pokud substituoval za anaforické proměnné, pak vrátí upravený `TilScript` objekt v odpovědi.

Pro performativ `query-ref` aplikace nejprve zkontroluje, zda hodnota property `ReplyBy`

v příchozí zprávě existuje a zda nevypršela lhůta, do kdy má na zprávu odpovědět. V případě, že lhůta ještě nevypršela, pak nejdříve vrátí notifikaci o svém záměru, zda dotaz zpracuje či nikoli. Agent se bude dotazem zabývat v případě, že důvěřuje odesílateli příchozí zprávy. To znamemá, že má jméno tohoto agenta uložené v databázi. Záměr je vyjádřen zprávou s performativem `agree` nebo `refuse`. Toto chování bylo implementována na základě FIPA specifikace pro dotazování mezi agenty (*SC00027H*).

Následně agent provede deserializaci TIL-Script XML a zpracování deserializovaného objektu pomocí zásuvných modulů. Namísto vrácení upraveného `TilScript` objektu zkusí porozumět obsahu dotazu. Aplikace dovede v současné implementaci porozumět jen dotazu na jména agentů, jaké má uložené v databázi. Agent na základě takového dotazu získá seznam jmen všech agentů a vygeneruje konstrukce v nichž tato jména poskytne. Všechny konstrukce jsou pak v rámci jediného TIL-Script XML odeslány v odpovědi.

Výstupy ze třídy `Decide` jsou tedy použity v odchozích zprávách. Před odesláním se vždy nejprve provede serializace obsahu zprávy (`Content`), nastaví se performativ a případně jazyk odchozí ACL zprávy.

Je nutno podotknout, že se jedná o minimální množství funkcionality nezbytné pro příjem a zasílání ACL zpráv. Odpovědi sestávají z konstrukcí, ve kterých bylo substituováno za anaforické proměnné nebo ze seznamu konstrukcí obsahujících jména agentů, jež jsou uložena v databázi.

6.7 Uživatelské rozhraní

Pro možnost demonstrovat functionalitu bylo implementováno uživatelské rozhraní. Při spuštění aplikace, respektive přístupu na webovou stránku, se zobrazí uvítací obrazovka a navigační menu. Navigovat se můžeme do tří různých formulářů pro práci s programem, anebo do sekce vytvořené za účelem prohlížení obsahu databáze.

První z formulářů *Receive* testuje situaci, kdy obdržíme nějakou ACL zprávu. Povinná pole ve formuláři jsou: *From* – adresa odesílatele, *To* – adresa příjemce a *Content* – obsah zprávy. Políčko s adresou příjemce je po načtení stránky automaticky vyplněno adresou, na jaké se agent nachází. Do pole *Content* se vkládá ACL zpráva obalující vnitřní zprávu v podobě TIL-Script XML. Po odeslání dojde ke zpracování zprávy programem včetně průchodu přes zásuvné moduly. Výstup se zobrazí v podobě ACL zprávy s odpovědí v poli *Output*. V příloze jsou k dispozici testovací ACL zprávy, na kterých lze tento formulář vyzkoušet.

Formulář *Send* poskytuje rozhraní pro testování odchozí komunikace. Sada povinných polí je stejná jako u prvního formuláře s tím rozdílem, že v políčku *From* se v tomto případě nachází adresa této aplikace a pole *To* obsahuje příjemce zprávy. Po odeslání formuláře dojde k přípravě a následnému poslání zprávy prostřednictvím rozhraní, které je nastaveno v konfiguraci (např. TCP socket).

Třetí v pořadí je formulář *Test plugins*, jenž lze využít pro testování zásuvných modulů. Formulář dynamicky vypisuje seznam všech zásuvných modulů a uživatel si může jednoduše

zaškrtnout, který modul nebo kombinaci modulů si chce vyzkoušet. Jak bylo zmíněno, výpis modulů je dynamický. Je-li přidán do systému nový modul, pak se tento seznam automaticky aktualizuje. Ve formuláři jsou dále dvě povinná políčka: *Conversation Id* – pro zadání identifikátoru konverzace a *XML File* pro nahrání TIL-Script XML souboru z disku. Tento formulář nepracuje s ACL zprávami, testuje práci přímo s moduly pro zpracování TIL-Script XML vstupů. Po odeslání formuláře dojde ke zpracování vstupů a na výstupu se v sekci *Output* vypíše seznam zpráv popisujících průchod jednotlivými moduly. V souvislosti s modulem pro řešení anafor se ku příkladu vypíše postup substituce, takže nejprve její zavedení a poté její provedení. Rovněž se na výstupu v části *Links* zobrazí seznam odkazů ke stažení XML souborů reprezentujících vstupy a výstupy jednotlivých modulů.

Všechny formuláře implementují základní validaci vstupů pomocí HTML5. Poslední zmíněný navíc provádí důkladnější validaci nahrávaného souboru. Kontroluje se například přípona souboru, obsah souboru, velikost a validace za použití XSD schémat. Tato jednotlivá omezení používají hodnoty, jež jsou definovány v konfiguraci.

Sekce k prohlížení obsahu databáze se nachází pod záložkou *Stored Data* a poskytuje rozcestník k jednotlivým tabulkám. Uživatelské rozhraní poskytuje výpis těchto tabulek: *Agents*, *Discourses* a *Discourse variables*. První zmíněná zobrazuje seznam všech agentů, jež TilMan zná. U každého agenta je uvedeno jméno, typ transportu a lokátor transportu, jenž agent používá jako primární. Není-li zobrazen typ transportu ani jeho lokátor, pak tuto informaci nemá TilMan k dispozici.

Druhá položka *Discourses* zobrazí všechny konverzace, jichž se agent účastnil. Na této stránce je k dispozici jednoduchý filtr pro vyhledávání podle hodnoty ve sloupci *Conversation Id*. Diskurzní proměnné jsou vypisovány pod položkou *Discourse variables*. Na této stránce je možnost filtrovat výpis podle sloupce *Discourse Id*. Tato hodnota se používá pro jednoznačnou identifikaci, do jakého diskurzu patří konkrétní diskurzní proměnná. Mimo to je možné výpis seřadit abecedně sestupně či vzestupně podle sloupců s hodnotou konstrukce (*Value*), konstruovaným datovým typem (*Entity Type*) či data poslední aktualizace diskurzní proměnné (*Last Time Updated*). Pro každý záznam si můžeme zobrazit detailní informace obsahující navíc například informaci, zda se jedná o kandidáta či nikoli.

Ve stránkách uživatelského rozhraní se používá technologie *Bootstrap*, což je open source soubor nástrojů pro vývoj s technologiemi HTML, CSS a JavaScript (JS). Díky této technologii je uživatelské rozhraní plně responzivní vůči velikosti obrazovky, na níž je zobrazeno. Pružně se tak přizpůsobí zobrazení ku příkladu na mobilních zařízeních.

Další technologií, s níž se ve stránce pracuje, je Asynchronous JavaScript And XML (AJAX), jež se používá pro komunikaci se serverem za použití HTTP požadavků. Skript využívající AJAX se vykonává na straně klientského prohlížeče a používá se pro odesílání a příjem informací v různých formátech. V tomto projektu je využito formátu JavaScript Object Notation (JSON), což je velice jednoduchý formát pro výměnu dat. Pro lidi je snadné jej číst i psát, pro stroje je snadné taková data parsovat a generovat. Prostřednictvím technologie AJAX jsou data odeslána

z prohlížeče klienta ve formátu JSON na pozadí.

V souvislosti s tímto způsobem předávání dat mezi klientem a serverem je namístě zmínit nekalou praktiku známou pod názvem *Cross-site request forgery* (také známá pod zkratkami XSRF a CSRF). Jedná se o útok proti aplikacím hostovaným na webu, kdy útočník může ovlivňovat interakci mezi prohlížečem klienta a webovou aplikací, jež danému prohlížeči důvěřuje. Tyto útoky jsou možné, jelikož prohlížeč automaticky zasílá nějaký druh autentizační známky, také nazývané *token*, s každým požadavkem. Jako prevenci před tímto útokem je ve frameworku ASP.NET Core od verze 2.0 do elementů formuláře stránky vždy vkládán tzv. *antiforgery token*. Tento token je unikátní a jeho hodnota se nadá předvídat. Díky tomu bude každé načtení formuláře (HTTP Get požadavek) pevně svázané s následným odesláním formuláře z prohlížeče klienta (HTTP Post požadavek) [15].

Další feature, kterou program využívá, je takzvaná *Session*. Tato vlastnost frameworku ASP.NET Core umožňuje ukládat data uživatele během doby, kdy si prohlíží webovou aplikaci. Sestává ze slovníku či hašovací tabulky uložené na serveru v paměti *cache* a uchovává informace o stavu session napříč požadavky zaslanými z prohlížeče. Díky tomu si aplikace po určitnou dobu pamatuje informace, které byly zadány do formulářů. V této aplikaci je doba nastavitelná v konfiguračním souboru.

Další informace o rozložení a používání uživatelského rozhraní jsou k dispozici v uživatelské dokumentaci.

Na závěr podkapitoly zmíním ještě webové API (Application Programming Interface), jež je v současné době k dispozici prostřednictvím formuláře *Receive*. Stejně jako jsou data odesílána na server pomocí AJAX technologie, lze využít libovolnou aplikaci, jež umí vytvářet HTTP požadavky. TilMan tedy umí přijímat zprávy nejen prostřednictvím TCP či UDP, ale také v podobě HTTP požadavků. Je však potřeba myslet na to, že ve zprávě musí být uveden XSRF token pro danou sekvenci Get a Post požadavků. Ve výpise 17 je příklad HTTP požadavku, jenž umí aplikace přijímat. Hodnoty v ostrých závorkách jsou pouze zástupné a skutečný požadavek musí obsahovat reálné hodnoty.

6.8 Testování

Při implementaci aplikace je obecně velmi důležité ověřit, zda jednotlivé části fungují tak, jak by měly. Pro kontrolu implementace některých metod proto byly definovány *unit testy*[27]. Všechny testy se nacházejí v projektu `TilMan.Tests` a jednotlivé třídy definující testy jsou vždy ukončeny sufixem `Tests`. K testování byl využit open source nástroj *xUnit*[21].

V projektu je otestováno několik oblastí. Mnohé definované testy sdílejí postup testu, kdy je z textového souboru načten vstup, se kterým se dále pracuje. Pokud je potřeba zkontrolovat složitější datovou strukturu, je v jiném textovém souboru uložen očekávaný výstup. Porovnání objektů se pak provádí pomocí tříd ve složce `Comparers`.

[27] Unit testy ověřují správnost implementace jednotky funkcionality (např. metody, procedury, apod.)

```
POST /Receive HTTP/1.1
Host: <TilMan URI>
Content-Type: multipart/form-data;
X-XSRF-TOKEN: <X-XSRF-TOKEN>

------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="From"

<Sender URI>
------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="To"

<TilMan URI>
------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="Content"

<message content>
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

Výpis 17: Příklad HTTP požadavku pro komunikaci s aplikací

Testy ve třídě `AclMessagingTests` kontrolují schopnost parsovat ACL zprávu. Ověřuje se, zda během parsování nedošlo k vyhození vyjímky.

Ve třídě `ContructionTests` se testuje funkcionalita metod třídy `Contruction` reprezentující konstrukci v TIL-Script XML datech. Pro příklad test s názvem `GetLambdaAbstraction` na základě načteného vstupu získá konstrukci, nad kterou provede lambda abstrakci od objektu (či objektů) typu daného konfigurací testu. Výsledek je porovnán s konstrukcí očekávaného výstupu. Porovnání se provádí pomocí třídy `ConstructionEqualityComparer` implementující metodu `Equals`. Dva objekty typu `Construction` jsou stejné, pokud hodnoty properties `Value`, `EntityType` a `Type` prvního objektu se shodují s hodnotami druhého objektu a také, pokud každá podkonstrukce prvního objektu se shoduje s podkonstrukcí druhého objektu. Toto ověření se provádí rekurzivně pro každou dvojici konstrukcí na stejné pozici ve stromové struktuře. Správná funkčnost metod pro definici a provedení substituce se ověřuje v testu `GetSubstitution`.

Metody objektu reprezentujícího TIL-Script XML se testují ve třídě `TilScriptTests`. Ku příkladu test `UpdateEntities` kontroluje, že je sekce `Entities` v TIL-Script XML doplněna o entity zmíněné v analyzovaných konstrukcích. Tato funkce se v aplikaci používá po aplikaci substituční metody, kdy se za anaforickou proměnnou dosadila konstrukce z diskurzu reprezentující nějakou entitu. Další testy jsou definovány ve třídách `ExpressionsTests` a `HelpersTests`.

V testech jsou použita ručně vytvořená data, která se nacházejí ve složce `Data` v projektu s testy a také jsou nahrána v příloze.

6.9 Konfigurace a nasazení

Aplikace se konfiguruje v textovém souboru `appsettings.json`, jenž je formátu JSON. V této podkapitole je uveden výčet jednotlivých položek v konfiguračním souboru, jejich výchozí hodnota a popis funkce.

1. `"AntiforgeryTokenName": "X-XSRF-TOKEN"` – název antiforgery tokenu, jenž se používá v HTTP požadavcích,
2. `"CleanupOlderThanDays": 7` – z databáze jsou při provádění úkolu čištění odstraněny záznamy starší daného počtu dní,
3. `"CleanupTaskCronExpr": "15 2 * * *"` – cron výraz, ve formátu minuta, hodina, den v měsíci, měsíc a den v týdnu, specifikuje, jak často se má provádět úkol čištění, výchozí hodnota specifikuje, že čištění se provádí každý den ve 2:15,
4. `"ConnectionStrings"` – definuje přístupové údaje pro databázi, používá se standardní formát zápisu pro SQL Server[22],
5. `"DiscourseCandidatesEnabled": true` – určuje, zda se po doplnění anafory ukládají výsledné konstrukce jako tzv. *kandidáti*,
6. `"DiscourseVariablesPerEntityType": 10` – definuje maximální počet diskurzních proměnných, které jsou uloženy pro jeden datový typ v daném diskurzu,
7. `"EntityTypesForLambdaAbstractionList": [ "Indiv" ]` – reprezentuje seznam datových typů, od kterých budeme abstrahovat při tvorbě diskurzu,
8. `"HostName": "localhost"` – název přiřazený danému zařízení v síti na němž aplikace běží,
9. `"LocalPort": 50000` – číslo portu, na kterém bude aplikace poslouchat, resp. přijímat příchozí TCP pakety,
10. `"Logging"` – slouží pro definici, jaké informace a z jakých úrovní aplikace jsou logovány,
11. `"MaximalUploadFileSize": 1048576` – maximální velikost souboru nahrávaného na server v bitech,
12. `"SaveXml": true` – nastavení, zda se před a po práci se vstupním XML souborem uloží XML data na disk pro možnost pozdější kontroly,
13. `"SessionIdleTimeoutTimeSpan": "0:20"` – délka session,
14. `"SessionName": ".TilMan.Session"` – název session,
15. `"PrivatePortStart": 49152` – začátek intervalu privátních portů,

16. `"PrivatePortEnd": 65535` – konec intervalu privátních portů,
17. `"ReceiveHttpRequest": "http://localhost:64887/Receive"` – adresa API pro příjem HTTP požadavků,
18. `"RootPath": "wwwroot"` – adresář pro ukládání statických souborů,
19. `"XsdPath": "schemas"` – název adresáře s XSD schématy,
20. `"XsdSchemas": [ "TILscriptXml.xsd" ]` – reprezentuje seznam XSD schémat, jež se použijí při validaci vstupních XML souborů.

Aby bylo možné pracovat s aplikací na lokálním počítači, je potřeba mít nainstalované prostředí pro sestavení a spuštění ASP.NET Core aplikací:

- .NET Core SDK[28]
- .NET Core Runtime[29]

Projekty využívají balíčky. Pro správnou funkci je proto potřeba mít v používaném IDE tyto balíčky nainstalované ve verzi uvedené v následujcím seznamu (případně v novějších verzích):

1. projekt TilMan
    - balíček Microsoft.AspNetCore.All verze 2.0.6
    - balíček Microsoft.NETCore.App verze 2.0.0
    - balíček Microsoft.VisualStudio.Web.CodeGeneration.Design verze 2.0.3
    - balíček NLog.Web.AspNetCore verze 4.5.2
    - balíček Hangfire verze 1.6.17
2. projekt TilMan.Tests
    - balíček Microsoft.NET.Test.Sdk verze 15.6.1
    - balíček Microsoft.NETCore.App verze 2.0.0
    - balíček xunit verze 2.4.0
    - balíček xunit.runner.visualstudio verze 2.4.0

Chceme-li aplikaci nasadit na serveru, je potřeba, aby podporoval framework ASP.NET Core ve verzi 2.0. Na serveru vytvoříme do příslušné složky dle specifikace webhostingu adresářovou strukturu do níž nahrajeme potřebné soubory. V příloze v adresáři `application/release` je

[28] `https://www.microsoft.com/net/download/Windows/build`
[29] `https://www.microsoft.com/net/download/Windows/run`

připravena aktuální verze aplikace pro nasazení. Dojde-li při dalším vývoji ke změnám ve zdrojových souborech, je potřeba provést `publish` nové verze a změněné soubory nahrát na server.

Po nasazení je vhodné provést konfiguraci aplikace v souboru `appsettings.json` a to především nastavení položek `HostName` a `ReceiveHttpRequest` dle místa, kam je aplikace nasazena. Pokud bude použita konkrétní databáze musí se nastavit hodnota `ConnectionStrings`. Rovněž je potřeba provést povolení portu, na němž aplikace poslouchá, na firewallu.

7 Použití a další možnosti rozšíření

V této kapitole budou uvedeny příklady použití agenta ve dvou situacích. Během první jsou agentu předávána data, z nichž si bude tvořit diskurz a také dosazovat za anaforické proměnné. Pro účely tohoto případu bude využit formulář *Test Plugins*. Výstupy budeme kontrolovat prostřednictvím stránek zobrazujících data z databáze. Ve druhé situaci je lokální agent dotázán vzdáleným agentem na jména agentů, které zná. Druhý příklad testuje také TCP komunikační rozhraní agenta.

Program jsem se snažila vyvíjet tak, aby jej bylo možné v budoucnu rozširovat. Podkapitola 7.3 pojednává o možných vylepšeních a rozšíření stávající funkcionality agenta. Zamýšlím se také v širším kontextu nad zasazením agenta do prostředí, v němž bude vykonávat svou činnost.

7.1 Případ 1: Řešení anafory

Agent se nachází v situaci, kdy na vstupu přijímá TIL-Script XML soubory obsahující analyzované konstrukce. Inspiraci jsem čerpala ze článku [7]. Případ sestává se čtyř analyzovaných konstrukcí:

1. „Bert is comming.“
2. „He is looking for a parking space.“
3. „So is Cecil.“
4. „He seeks him.“

S každým přijatým souborem agent provede aktualizaci diskurzu a také substituci za anaforické proměnné, je-li to možné. Pro názornost jsou jednotlivé kroky zachyceny ve výpisech zdrojového kódu. V rámci jednoho kroku je vždy uvedena konstrukce v notaci TIL-Script vyjadřující daný výraz. Při každé aktualizaci diskurzu je vypsán seznam konstrukcí uložených do diskurzu včetně typu objektu, který konstruují. Pokud se jednalo o otevřenou konstrukci, pak je uveden zápis konstrukce po definici substituce a také po provedení substituce za anaforickou proměnnou. Byla-li v rámci substituce dosazena konstrukce ve tvaru lambda abstrakce, pokusí se agent provést omezenou β-redukci jménem a je vypsán výsledek této operace. Po substituci se uvádí i seznam konstrukcí uložených do diskurzu s příznakem kandidát. Každý výpis je opatřen legendou s popisem, co jednotlivé řádky reprezentují. Výstupy pro první dva výrazy jsou zobrazeny ve Výpisech 18 a 19.

V následujícím kroku agent provede několik akcí navíc, které však nejsou zachyceny ve Výpise 20. Po obdržení nové konstrukce si agent ověřil, že se její hodnota liší oproti konstrukci, se kterou pracoval v předchozím kroku. Agent tomu rozumí tak, že iniciátor komunikace byl spokojen s dosazením dané diskurzní proměnné do konstrukce a nepožaduje další konstrukci s dosazením následující diskurzní proměnné. V takové situaci dochází k uložení kandidátů.

```
1  [\w:World [\t:Time [[['Coming w] t] 'Bert]]]
2
3  'Coming -> (((Bool Indiv) Time) World)
4  'Bert -> Indiv
5  [\w:World [\t:Time [[['Coming w] t] 'Bert]]] -> ((Bool Time) World)
6  [\w:World [\t:Time [\x:Indiv [[['Coming w] t] x]]]] -> (((Bool Indiv) Time) World)
```

1	analyzovaná konstrukce
3 – 6	aktualizace diskurzu

Výpis 18: Případ 1: Řešení anafory – „Bert is comming.“

```
1  [\w:World [\t:Time [[['Looking_for w] t] he 'Park_Space]]]
2
3  'Looking_for -> (((Bool Indiv (((Bool Indiv) Time) World)) Time) World)
4  'Park_Space -> (((Bool Indiv) Time) World)
5
6  he -> Indiv
7  'Bert -> Indiv
8
9  [\w:World [\t:Time ^2['Sub ''Bert 'he '[[['Looking_for w] t] he 'Park_Space]]]]
10 [\w:World [\t:Time [[['Looking_for w] t] 'Bert 'Park_Space]]]
11
12 [\w:World [\t:Time [[['Looking_for w] t] 'Bert 'Park_Space]]] -> ((Bool Time) World
       )
13 [\w:World [\t:Time [\x:Indiv [[['Looking_for w] t] x 'Park_Space]]]] -> (((Bool
       Indiv) Time) World)
```

1	analyzovaná konstrukce
3 – 4	aktualizace diskurzu před substitucí
6	anaforická proměnná, jež se v konstrukci vyskytuje
7	nalezená diskurzní proměnná
9	definice substituce
10	aplikace substituce
12 – 13	aktualizace diskurzu po substitutci (kandidáti)

Výpis 19: Případ 1: Řešení anafory – „He is looking for a parking space.“

Druhou akcí agenta je rozpoznání, že diskurzní proměnná substituovaná do konstrukce byla ve tvaru lambda abstrakce. Zároveň ví, že pokud takovou konstrukci dosazoval, pak se může pokusit aplikovat argument (příp. argumenty) na funkci za použití omezené β-redukce. Pokud jsou tedy argumenty typu Proměnná či Trivializace atomické entity, pak může omezenou β-redukci provést. V souvislosti s Výpisem 20 to znamená, že po provedení kroku 9 agent zkontroloval dosazenou konstrukci a její argument.

```
1  [\w:World [\t:Time [[[So w] t] 'Cecil]]]
2
3  'Cecil -> Indiv
4
5  So -> (((Bool Indiv) Time) World)
6  [\w:World [\t:Time [\x:Indiv [[['Looking_for w] t] x 'Park_Space]]]] -> (((Bool
       Indiv) Time) World)
7
8  [\w:World [\t:Time ^2['Sub '[\w:World [\t:Time [\x:Indiv [[['Looking_for w] t] x '
       Park_Space]]]] 'So '[[[So w] t] 'Cecil]]]]
9  [\w:World [\t:Time [[[[\w:World [\t:Time [\x:Indiv [[['Looking_for w] t] x '
       Park_Space]]]] w] t] 'Cecil]]]
10
11 [\w:World [\t:Time [[['Looking_for w] t] 'Cecil 'Park_Space]]]
12
13 [\w:World [\t:Time [[['Looking_for w] t] 'Cecil 'Park_Space]]] -> ((Bool Time)
       World)
```

1	analyzovaná konstrukce
3	aktualizace diskurzu před substitucí
5	anaforická proměnná, jež se v konstrukci vyskytuje
6	nalezená diskurzní proměnná
8	definice substituce
9	aplikace substituce
11	konstrukce po provedení omezené β-redukce
13	aktualizace diskurzu po substitutci (kandidáti)

Výpis 20: Případ 1: Řešení anafory – „So is Cecil.“

Nyní si vyzkoušíme, jak je agent schopen dosazovat do konstrukce s více anaforickými proměnnými. Očekávaným výsledkem je postupné dodání všech možných kombinací diskurzních proměnných, jež má agent uložené v databázi. Použijeme konstrukci konstruující výraz „He seeks him.“. Aktuálně by diskurz měl obsahovat dvě položky pro typ `Indiv` a to konstrukce `'Cecil` a `'Bert`. Výpis 21 se poněkud liší od předchozích tím, že agentu v této chvíli posíláme pětkrát za sebou stejnou analyzovanou konstrukci a zobrazujeme jen výsledné konstrukce po provedení substituce. Při pátém pokusu již žádná další možná kombinace neexistuje, modul pro řešení anafory v takovém případě vrátí nezměněnou konstrukci, kterou obdržel na vstupu.

```
1  [\w:World [\t:Time [[['Seek w] t] he him]]]
2
3  [\w:World [\t:Time [[['Seek w] t] 'Cecil 'Cecil]]]
4  [\w:World [\t:Time [[['Seek w] t] 'Cecil 'Bert]]]
5  [\w:World [\t:Time [[['Seek w] t] 'Bert 'Cecil]]]
6  [\w:World [\t:Time [[['Seek w] t] 'Bert 'Bert]]]
```

Výpis 21: Případ 1: Řešení anafory – „He seeks him.“

7.2 Případ 2: Dotazování na jména agentů

V tomto případu figurují dva agenty A a B, přičemž agent B se dotazuje agenta A na jména agentů, které agent A zná. Komunikace probíhá prostřednictvím protokolu TCP. Pro iniciaci komunikace agentem B využijeme formulář *Send* z uživatelského rozhraní. Scénář zahrnuje:

1. Zaslání dotazu agentem B,
2. odmítnutí poskytnutí údajů agentem A z důvodu, že nezná agenta B,
3. zaslání *inform* zprávy agentem B, čímž dojde k zaregistrování B u agenta A,
4. opakování zaslání dotazu agentem B,
5. agent A poskytne požadované údaje.

Dotaz zasílaný agentem B je vyobrazen ve Výpise 22, obsah zprávy byl však ručně upraven. V reálné zprávě dotaz obsahuje TIL-Script XML zprávu, pro přehlednost je zde však uveden pouze její TIL-Script ekvivalent. Všimněme si nyní identifikátoru konverzace. Pomocí tohoto identifikátoru si můžeme v sekci *Stored Data* uživatelského rozhraní ověřit, že požadavek byl odmítnut a to tak, že do diskurzu agenta B nebyla pro danou konverzaci uložena žádná informace relevantní pro zaslaný dotaz. Následně zasílá agent B *inform* zprávu. Obsah zprávy není pro tento případ nijak důležitý, je však podstatné, že na základě této zprávy si agent A zaregistuje jméno agenta B. Nyní může agent B zopakovat dotaz z Výpisu 22. Obdržení odpovědi si zkontrolujeme v sekci *Stored Data*. Na Obrázku 12 si můžeme prohlédnout obrazovku uživatelského rozhraní obsahující požadované informace.

```
(query-ref
  :sender (agent-identifier :name 10.0.0.9:49152)
  :receiver (set (agent-identifier :name 10.0.0.9:50000))
  :content
    "[\w:World [\t:Time [\x:Indiv [['Agent w] t] x]]]"
  :language TilScript
  :conversation-id cf4fff4f-ed38-ff6e-e861-f752f283e86b)
```

Výpis 22: Případ 2: Dotazování na jména agentů – ACL zpráva s dotazem

Discourse variable

Back to full List

Filter by: 13017 Search

Discourse Id	Value	Entity Type	Last Time Updated	Type	
13017	'Agent	(((Bool Indiv) Time) World)	2018-04-18 20:25:05	Trivialisation	Details
13017	'Bert	Indiv	2018-04-18 20:25:05	Trivialisation	Details
13017	[w:World [t:Time [[x:Indiv [[['Agent w] t] x]] 'Bert]]]	(((Bool Indiv) Time) World)	2018-04-18 20:25:05	Closure	Details
13017	[w:World [t:Time [y:Indiv [[x:Indiv [[['Agent w] t] x]] y]]]]	(((Bool Indiv) Time) World)	2018-04-18 20:25:05	Closure	Details
13017	'Coming	(((Bool Indiv) Time) World)	2018-04-18 20:23:43	Trivialisation	Details
13017	[w:World [t:Time [[['Coming w] t] 'Bert]]]	((Bool Time) World)	2018-04-18 20:23:43	Closure	Details
13017	[w:World [t:Time [x:Indiv [[['Coming w] t] x]]]]	(((Bool Indiv) Time) World)	2018-04-18 20:23:43	Closure	Details

Obrázek 12: Případ 2: Dotazování na jména agentů – ověření přijetí odpovědi na dotaz

7.3 Další možnosti vylepšení a rozšíření

Jednou z užitečných dovedností agenta je definice a práce s vlastní bází znalostí. Do báze si může agent ukládat různá fakta zjištěná během konverzací s jinými agenty, provádět nad nimi různé logické úvahy, apod. Modul pro řešení anafory by zajisté pomohl při využívání báze znalostí. Bez báze znalostí může agent jen těžko porozumět významu zpráv, s nimiž pracuje. Kromě lokální báze může být agentu užitečné i napojení na online ontologickou bázi znalostí (např. lexikální báze typu WordNet [23]), z níž by mohl agent získat data pro analýzu výrazů přirozeného jazyka. Agentu by se při práci s ontologickou bází mohlo hodit i ukládání celých konverzací do databáze. Měl by tak možnost následně nad historií konverzace provádět usuzování. Agent by dále mohl být schopen odvozovat souvislosti ze získávaných informací, ku příkladu „kdo se ptal na agenta x“.

Nynější implementace algoritmu počítá s tím, že vstupní data již prošla typovou analýzou. To znamená, že všechny konstrukce mají přiřazen datový typ podle toho, jaký objekt konstruují. Algoritmus by zajisté působil sofistikovaněji, pokud by uměl vhodně odhadnout datový typ objektu konstruovaného danou konstrukcí. Jednou z možností je vycházet z vlastní báze znalostí. Například by agent věděl, že řídit motová vozidla mohou v pouze individua. Po přijetí zprávy obsahující výraz „(On) Řídí auto.“ by dovedl, i bez předchozí analýzy volné proměnné *on*, dosadit vhodnou diskurzní proměnnou. Případně, pokud by určení datového typu nebylo jednoznačné, by agent mohl zkoušet dosazovat různé atomické typy.

Při práci vyvstala situace, kdy agent obdržel zprávu obsahující anaforickou proměnnou „já“. Na této anafoře je zajímavé, že jen zřídka bývá v předchozí konverzaci zmíněn antecedent této proměnné. Stojí jistě za úvahu analyzovat tuto skutečnost a navrhnout možnost, jak elegantně za takovou proměnnou dosazovat jméno konkrétního agenta bez potřeby prohledávat diskurzní proměnné.

Z pohledu implementace zmíním poněkud nesnadnou práci s datovými typy v podobě řetězců. Velmi užitečná a méně náchylná k chybám by byla reprezentace datového typu v podobě objektu. Textový řetězec datového typu může sestávat z jediného slova, ale u některých funkcí může jít i o poměrně komplikovanou strukturu. Vhodná by byla ku příkladu reprezentace v podobě stromu. Díky tomuto přístupu by se velmi snadno prováděla také typová kontrola konstrukcí.

Agent umí přijímat zprávy prostřednictvím HTTP požadavků. Dalším možným rozšířením je implementace třídy pro odesílání HTTP požadavků jiným agentům. Princip uchovávání adres agentů je k tomuto uzpůsoben. Bylo by však potřeba upravit metody zpracovávající ACL zprávy, aby v případě, kdy agent preferuje příjem zpráv přes HTTP protokol, bylo možné tohoto kanálu využít. V souvislosti s HTTP požadavky jsem si vědoma neduhu, kterým je použití absolutní adresy API pro příjem HTTP požadavků ve třídě `HttpRequestHelper`. Tento nedostatek by bylo možné vyřešit použitím jiného přístupu pro předání přijaté TCP zprávy uvnitř agenta, čímž by došlo ke zrušení položek `HostName` a `ReceiveHttpRequest` z konfiguračního souboru a nebylo by potřeba je s každým novým nasazením agenta nastavovat.

Dle FIPA specifikace by agent, který neporozumí významu obdržené zprávy, měl odpovědět zprávou s performativem *Not understood*. Část této funkcionality je v současné době implementována. Například, pokud dojde k chybě při deserializaci XML vstupu, posílá se odpověď s tímto performativem zpět odesílateli. Je ovšem otázka, jak se má agent zachovat v situaci, kdy samotná ACL zpráva má chybnou syntaxi. Aktuálně je sice zachycena výjimka, jež dovede velmi dobře navést ke konkrétní nepřesnosti v syntaxi, nicméně nedochází k propagaci informace zpět odesílateli. Bylo by potřeba se zamyslet nad formátem obsahu takové zprávy, odesílateli by však taková informace mohla velmi pomoci.

V multi-agentním systému je také potřeba, aby existoval nějaký centrální bod, kterého se agenty mohou doptávat na adresy jiných agentů, tzv. *resolver*. Dalším možným rozšířením je tedy implementace tohoto bodu. Případně možnost, aby touto schopností disponovaly agenty v systému. Mohly by se pak navzájem dotazovat na adresy jiných agentů. Jednalo by se o emergentní skupinovou znalost v multi-agentním systému.

Rozhodovací modul je připraven pro další možnosti rozšiřování, ať už přidáním dalších performativů, s nimiž by agent uměl pracovat, nebo rozšířením funkcionality existujících performativů. Agent je v aktuální implementaci reaktivní v tom smyslu, že sám nikdy neiniciuje konverzaci, pouze odpovídá na obdržené zprávy. Pokud by se našel nějaký rozumný důvod, mohl by být agentu upraven rozhodovací modul tím způsobem, že by dovedl z databáze získat adresu konkrétního agenta a zahájit s ním konverzaci. Ku příkladu by mohl agent být schopen získat od uživatele seznam úkolů s termínem splnění, například vyhledat nějaké informace na internetu nebo rezervovat letenky. V takovém případě by během plnění úkolu zahajoval nové konverzace s jinými agenty. Pak by ovšem rovněž dávalo smysl, aby si agent aktualizoval diskurz také zprávami, které by posílal on sám.

Při práci s gramatikou TIL-Scriptu jsem si všimla, že gramatika zahrnuje také definici bílých znaků (`optional whitespace`), nepřišla jsem však na valný přínos definice bílých znaků v gramatice. Během práce jsem si vyzkoušela napsat lexikální a syntaktický analyzátor a zjistila jsem, že při tomto přístupu jsou bíle znaky při parsování ignorovány. Osobně vidím možnost zjednodušení gramatiky TIL-Script vypuštěním definice bílých znaků.

8 Závěr

V práci jsou popsány teoretické základy pro analýzu výrazů přirozeného jazyka pomocí formálního systému TIL, se zaměřením na dynamické zpracování anaforických odkazů. Rovněž je zde uvedena definice a popis jazyka TIL-Script, komputační varianty systému TIL. Tento jazyk slouží k popisu objektů reálného světa konstrukcemi (procedurami).

Dále byl navržen a implemenován algoritmus pro zpracování dynamického diskurzu v jazyce C#. Tento algoritmus byl zasazen formou zásuvného modulu do softwarového agenta, který je schopný komunikovat v jazyce ACL s dalšími agenty v multi-agentním systému. Agenty si mohou předávat zprávy za použití protokolu TCP či HTTP a tyto zprávy obsahují analyzované konstrukce ve formátu XML.

Agent si během konverzace s jinými agenty automaticky buduje seznam konstrukcí o nichž se mluvilo, přičemž data jsou uložena v databázi. Pro potřeby otestování funkcionality agenta byla ručně vytvořena vstupní data, jejichž správná struktura je ověřována XSD schématem. S využitím těchto dat byl algoritmus otestován pomocí Unit testů.

Na závěr textu jsou uvedeny případy užití agenta ve dvou konkrétních situacích, a popsány možnosti, jak lze aplikaci dále rozšířit.

Literatura

[1] TICHÝ, Pavel. *The Foundations of Frege's Logic.* New York: de Gruyter, 1988. ISBN 3110116685.

[2] DUŽÍ, Marie a Pavel MATERNA. *TIL jako procedurální logika: průvodce zvídavého čtenáře Transparentní intensionální logikou.* Bratislava: Aleph, 2012. ISBN 9788089491087.

[3] DUŽÍ, Marie, Bjørn JESPERSEN a Pavel MATERNA. *Procedural semantics for hyperintensional logic: foundations and applications of transparent intensional logic.* New York: Springer, c2010. ISBN 9789048188116.

[4] PIERCE, Benjamin C. *Types and programming languages.* Cambridge, Mass.: MIT Press, 2002. ISBN 9780262162098.

[5] THOMPSON, Simon. *Type theory and functional programming.* Reading, Mass.: Addison-Wesley, c1991. ISBN 0201416670.

[6] DUŽÍ, Marie a Miloš KOSTEREC. A valid rule of β-conversion for the logic of partial functions. *Organon F.* 2017, **24**(1), 10–36. ISSN 1335-0668.

[7] DUŽÍ, Marie. Logic of Dynamic Discourse; Anaphora Resolution. *Frontiers in Artificial Intelligence and Applications*, vol. 301: Information Modelling and Knowledge Bases XXIX, pp. 263-279, Amsterdam: IOS Press.

[8] *The Foundation for Intelligent Physical Agents* [online]. Washington, D.C., c2018 [cit. 2018-04-20]. Dostupné z: `http://www.fipa.org/`

[9] *The ASP.NET Site* [online]. c2018 [cit. 2018-04-20]. Dostupné z: `https://www.asp.net/`

[10] *Overview of ASP.NET Core MVC* [online]. c2018 [cit. 2018-04-20]. Dostupné z: `https://docs.microsoft.com/en-us/aspnet/core/mvc/overview`

[11] VERMA, Rishabh, a Neha SHRIVASTAVA. *.NET Core 2.0 By Example: Learn to program in C# and .NET Core by building a series of practical, cross-platform projects.* Birmingham: Packt Publishing, 2018. ISBN 1788390261.

[12] *Introduction to ASP.NET Core* [online]. c2018 [cit. 2018-04-20]. Dostupné z: `https://docs.microsoft.com/en-us/aspnet/core/`

[13] *Dependency Inversion Principle* [online]. c2018 [cit. 2018-04-20]. Dostupné z: `http://deviq.com/dependency-inversion-principle/`

[14] *Building simple plug-ins system for ASP.NET Core* [online]. c2017 [cit. 2018-04-20]. Dostupné z: `http://gunnarpeipman.com/2017/01/aspnet-core-plugins/`

[15] *Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core* [online]. c2017 [cit. 2018-04-20]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.1

[16] *Hangfire – Background jobs and workers for .NET and .NET Core* [online]. c2017 [cit. 2018-04-20]. Dostupné z: https://www.hangfire.io/

[17] *GitHub - NLog/NLog: NLog - Advanced and Structured Logging for Various .NET Platforms* [online]. c2018 [cit. 2018-04-20]. Dostupné z: https://github.com/NLog/NLog

[18] *Asynchronous Server Socket Example* [online]. c2017 [cit. 2018-04-20]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/network-programming/asynchronous-server-socket-example

[19] *Service Name and Transport Protocol Port Number Registry* [online]. c2018 [cit. 2018-04-20]. Dostupné z: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[20] *A More Efficient Regex Tokenizer – Jack Vanlightly* [online]. c2016 [cit. 2018-04-20]. Dostupné z: https://jack-vanlightly.com/blog/2016/2/24/a-more-efficient-regex-tokenizer

[21] *About xUnit.net* [online]. c2017 [cit. 2018-04-20]. Dostupné z: https://xunit.github.io/

[22] *SQL Server connection strings* [online]. c2016 [cit. 2018-04-20]. Dostupné z: https://www.connectionstrings.com/sql-server/

[23] *WordNet. A Lexical Database for English* [online]. c2018 [cit. 2018-04-20]. Dostupné z: https://wordnet.princeton.edu/

A Obsah DVD

Veškeré přílohy jsou uloženy na přiloženém DVD. Jednotlivé složky obsahují:

- `application` – vlastní aplikace
  - `release` – ZIP archív se soubory potřebnými pro nasazení aplikace na serveru
  - `source` – ZIP archív s kompletním projektem z programu Visual Studio 2017
- `diploma` – text diplomové práce
  - PDF verze
  - `source` – zdrojové soubory pro LaTeX včetně použitých obrázků a Makefile
- `documentation`
  - `programmer-manual` – programátorská příručka vygenerovaná programem Doxygen z komentářů ve zdrojovém kódu aplikace
    - `html.zip` – ZIP archív HTML verze
  - `user-manual` – uživatelská příručka
    - PDF verze
    - `source` – zdrojové soubory pro LaTeX včetně použitých obrázků a Makefile
- `examples` – XML soubory s ručně vytvořenými testovacími příklady
  - `case-study` – data pro případy použití
  - `testing-data`
    - `manual` – data pro manuální testování
    - `unit-tests` – data použitá v aplikaci při unit testech
  - `TILscriptXml.xsd` – XSD schéma pro validaci TIL-Script XML souborů
- `materials` – související materiály, které mi poskytla vedoucí práce
  - `Agent.zip` – zdrojové kódy původní aplikace agenta
  - `example of TIL-Script message.tils` – příklad TIL-Script zprávy
  - `example of TIL-Script message.xml` – příklad TIL-Script XML zprávy
  - `TIL-Script grammar ebnf v5.1.txt` – gramatika jazyku TIL-Script